Semantic Knowledge Representation and Analysis

by

Dina Kachintseva

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2011

Certified by .....
Dr. David Brock
Principal Research Scientist
Thesis Supervisor

Accepted by ..
Dennis M. Freeman
Chairman, Department Committee on Masters of Engineering Theses

Semantic Knowledge Representation and Analysis

by

Dina Kachintseva

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2011, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Natural language is the means through which humans convey meaning to each other - each word or phrase is a label, or name, for an internal representation of a concept. This internal representation is built up from repeated exposure to particular examples, or instances, of a concept. The way in which we learn that a particular entity in our environment is a "bird" comes from seeing countless examples of different kinds of birds, and combining these experiences to form a mental representation of the concept. Consequently, each individual's understanding of a concept is slightly different, depending on their experiences. A person living in a place where the predominant types of birds are ostriches and emus will have a different representation birds than a person who predominantly sees penguins, even if the two people speak the same language.

This thesis presents a semantic knowledge representation that incorporates this fuzziness and context-dependence of concepts. In particular, this thesis provides several algorithms for learning the meaning behind text by using a dataset of experiences to build up an internal representation of the underlying concepts. Furthermore, several methods are proposed for learning new concepts by discovering patterns in the dataset and using them to compile representations for unnamed ideas. Essentially, these methods learn new concepts without knowing the particular label - or word - used to refer to them.

Words are not the only way in which experiences can be described - numbers can often communicate a situation more precisely than words. In fact, many qualitative concepts can be characterized using a set of numeric values. For instance, the qualitative concepts of "young" or "strong" can be characterized using a range of ages or strengths that are equally context-specific and fuzzy. A young adult corresponds to a different range of ages from a young child or a young puppy. By examining the sorts of numeric values that are associated with a particular word in a given context, a person can build up an understanding of the concept. This thesis presents algorithms that use a combination of qualitative and numeric data to learn the meanings of concepts. Ultimately, this thesis demonstrates that this combination of qualitative and quantitative data enables more accurate and precise learning of concepts.

Thesis Supervisor: Dr. David Brock
Title: Principal Research Scientist

Acknowledgments

I would like to thank David Brock for giving me the opportunity to work on this project and for his constant support and guidance throughout. I am very grateful for the amount of time and effort he has invested over the past two years to answer any questions I had, to discuss every aspect of my thesis and to patiently read through my rough drafts.

I would like to thank my parents and brother for providing me with every possible kind of support. Starting with getting into MIT, through the last couple weeks of my thesis writing, your love and help were there for me every step of the way, making all of this possible.

Finally, I would like to thank my friends for making my life over the past five years at MIT full of fun, caring support and wonderful memories. I would not have been able to do this without you.

Contents

List of Figures

Chapter 1

Introduction

Capturing and representing the meaning behind natural language has been a long-standing goal in computer science. Over the years, many knowledge representations have been developed, from strict logical function definitions in λ-calculus, to more flexible Semantic Web ontologies, to semantic networks such as MultiNet. The goal of these schemes is to capture the semantic relationships and entities described in text. One of the inherent difficulties of this task is that the "meaning" of a word is a fluid concept that depends on the context in which it is used and often has many nuances and connotations. Thus, any sort of formal logic to define an entity or relationship has to accommodate this variability and fuzziness in meaning. Furthermore, since relationships and entities are based on underlying rules, these schemes require a way to represent them. These defined rules then allow further inference and reasoning.

We propose several algorithms for detecting the relationships and patterns that exist in semantic datasets. In doing so, we believe that the meaning of any given word or concept is context-dependent. Thus, there is no universal single "meaning" for a given concept. Rather, there is a probabilistic "prototype" describing what attributes this concept likely has in a given context. The word "young", for instance, when used to describe a human infant, has a different meaning from when it is used to describe a young adult. Similarly, the word would have different meanings when applied to human beings in general, depending on the time period in human history being discussed. Furthermore, we believe that existing semantic network representations do not put enough of an emphasis on the numeric components of concepts. Often, events or entities are described using quantitative measures, such as height, width, duration, speed, strength, and many others. This quantitative data contains a wealth of information that can be used to learn more about the meaning of a concept, given the right algorithms. Many purely qualitative ideas can be described and inferred from numeric data. We propose several techniques for detecting numeric patterns and combining them with qualitative attributes to both (1) obtain a more complete and informative representation of concepts (2) learn

new concepts.

1.1 Motivations for Semantic Knowledge Representation

The ability to represent the concepts described by natural language makes it possible for computers to perform tasks that currently require human expertise. Some of these include:

- **Entity Extraction** - Given a piece of text, retrieve all information or attributes of a particular person, organization, event, or any other specified entity. This task is currently most often performed using statistical natural language processing techniques, which provide no underlying understanding of the text. These algorithms use measures of word frequency and distance to detect relevant words and phrases.
- **Matching Text On Meaning** - Given several pieces of text, determine which two match most closely based on underlying meaning. These texts may describe the same, or similar concepts, using different vocabulary. Thus, this task requires an understanding of which words are semantically similar.
- **Modeling and Prediction** - Given some observed set of events, scenarios, sentences or any other type of entity, identify underlying semantic patterns; then apply these patterns to new scenarios to model and predict their characteristics.

1.1.1 Modeling and Prediction

Semantic modeling and prediction encompasses a wide range of tasks.

At a sentence scope, we can detect a semantic pattern for a particular set of sentences and then use it to suggest how to complete a new sentence. For example, if we see a series of sentences such as:

- John wrote a story.
- Jane typed up an email.
- Jim scribbled a note.

we would infer that there is an overarching semantic pattern: (1) *Name* (2) *method of writing* (3) *body of text.* We could then predict, given any other sentence that starts with a name and a method of writing, that it likely ends with a body of text.

At a scenario scope, we can detect semantic patterns for a series of events and use this to predict the likely progression of events for similar scenarios. For example, if we were to encapsulate the semantic event pattern in the cartoon Wiley Coyote and the Roadrunner, it would probably look

something like this: (1) *Wiley Coyote comes up with plan to catch Roadrunner.* (2) *Roadrunner evades plan, causes it to malfunction.* (3) *Wiley Coyote gets hurt; Roadrunner gets away.* We could then use this pattern to predict the sequence of events and outcome of every single cartoon in this series. Similarly, if we examine a scenario involving a natural disaster, we would observe that different sequences of events lead to different outcomes. For example, in the event of a severe tornado, we would expect building destruction, possible fatalities, power outages, and, ultimately, an increase in the amount of insurance claims. In the event of a weak tornado, we might expect to see minor building damage, no fatalities or power outages and a much smalled amount of insurance claims. Given more specific attributes of the tornado, we could predict with higher accuracy the type and severity of the outcomes we are likely to observe. Depending on the content and level of detail of a given dataset, many different semantic patterns can be detected and leveraged.

1.2 Thesis Summary

This thesis follows the progression shown below:

- Chapter 2 gives an overview of several past semantic knowledge representation attempts and how they differ from our representation.
- Chapter 3 describes our semantic knowledge representation.
- Chapter 4 outlines our application, which implements the knowledge representation.
- Chapter 5 explains the algorithms we developed to discover semantic patterns and learn the meanings of concepts in our knowledge representation.
- Chapter 6 describes the datasets built to test these algorithms.
- Chapter 7 discusses the results obtained by running our algorithms on the datasets.
- Chapter 8 suggests areas of potential improvement and further work.

Chapter 2

Background

Numerous attempts have been made to capture and represent human knowledge and language in a format that is machine-readable. Many approaches have focused on natural language processing techniques, using various statistical models to infer the importance and roles of words in a body of text. On the other end, there are examples such as the Semantic Web, which attempt to capture the meaning of language through human-constructed semantic ontologies, which attempt to define the roles of words through logical rules. Finally, there are numerous existing semantic knowledge networks which represent concepts as nodes and relationships between them as links. These sorts of networks often treat the definition of a concept or relationship as an objective truth, rather than a subjective, context-dependent set of characteristics. Furthermore, these networks focus primarily on qualitative (and not quantitative) attributes to represent and learn about concepts.

2.1 Statistical Natural Language Processing

One of the main strategies used in natural language is to use statistical inference as a way to extrapolate semantic labels, rules, or attributes for the underlying words and concepts. Although there are many different kinds of statistical natural language processing algorithms, most use a combination of large text corpuses to train their algorithms and word weighting schemes that are based on word frequency or distance between words.

Among the most commonly and successfully used word weighting schemes is *inverse document frequency (IDF)* [8]. The IDF for a given word is calculated by dividing the occurrence frequency of the word within a specific text by the total frequency of the word in all known texts. This measure makes it easier to distinguish important content words from common semantically unimportant words. For example, given a text on physics, words such as *the*, or *a* would be assigned low IDFs while context-specific words such as *centripetal* would be assigned high IDFs.

Another example of natural language statistical inference is *latent semantic analysis (LSA)*. This

technique makes the assumption that when two words are close in meaning, they will occur close together in the text. LSA builds a matrix which contains word counts per paragraph from a large piece of text. It then uses a technique called singular value decomposition (SVD) to find the words that are similar to one another by mathematically manipulating the matrix.

While these sorts of statistical semantic natural language processing mechanisms can provide baseline weights for words, and can be used to gain insight into the semantic makeup of a piece of text, they do not represent the inherent meaning of the text.

2.2 The Semantic Web

The Semantic Web is a framework that attempts to capture and represent the information on the World Wide Web in a semantically meaningful way. It has been described by its creator Tim Berners-Lee as "a web of data that can be processed directly and indirectly by machines" [10].

The Resource Description Framework (RDF) provides the syntax for storing Semantic Web data in the form of "triples". Each triple consists of three parts: subject, predicate, object. The predicate defines the relationship between the subject and the object. For instance:

Subject: Brad Pitt **Predicate:** Type Of **Object:** Actor

Subject: Jerry Bruckheimer **Predicate:** Producer Of **Object:** Remember the Titans

For each type of predicate, the Semantic Web defines a set of rules using the Ontology Web Language (OWL) that restrict the scope of the subject and object that this predicate can operate on. OWL also defines the class hierarchy of all concepts that can fill the subject or object roles. As a result, the Semantic Web is often represented as a connected graph, where the edges are the predicates and the nodes are the subject and object values. Among the numerous existing predicates, two of the most commonly used ones denote the type-of and subclass relationships: rdf:type and rdfs:SubClassOf. As described in the *RDF Primer*, the rdf:type predicate is used to establish the relationship between a class and an instance of that class, while the rdfs:SubClassOf is used to build a class hierarchy and define the relationship between a parent and child class [9].

A single entity on the Semantic Web can have multiple RDF types, and a single class can be a subclass of multiple classes. Furthermore, anyone can define a new class, predicate, ontology and set of rules to suit their needs and their specific dataset. This leads to data that can be at the same time redundant and inconsistent, as people develop overlapping or conflicting sets of definitions. For instance, if we take a look at the RDF dataset that corresponds to Wikipedia knowledge - known as DBPedia - we can see that *New York City* is a type of all of the following classes:

```
PortCitiesInTheUnitedStates
FormerCapitalsOfTheUnitedStates
SettlementsEstablishedIn1625
Thing
```

```
PortCity
USCity
Village108672738
FormerUnitedStatesStateCapitals
CoastalCitiesInTheUnitedStates
CitiesInNewYork
```

Interestingly enough, although many of these categories seem like they would belong in the same hierarchy - for instance, PortCity and FormerCapitalsOfTheUnitedStates should both be types of cities, or PortCitiesInTheUnitedStates should be a subclass of PortCity, or CitiesInNewYork should be a subclass of USCity - this is not always the case. Furthermore, perhaps the most strikingly flawed aspect of these types is that the name of each contains valuable semantic information that cannot be extracted or connected to other existing concepts. The above categories reveal a number of facts about New York City, including that it is in the United States, it is in New York and it was established in 1625. Just as importantly, it is not possible to infer that New York is a coastal city, because the hierarchy of CoastalCitiesInTheUnitedStates does not derive from a category CoastalCities, as would seem logical.

All these discrepancies illustrate that it is not enough to define classes by simply the hierarchies they belong to. It is just as vital to be able to associate a class with a set of attributes. Furthermore, it would be highly useful to recognize similarities between classes that share attributes, identifying them as either equivalent or at least correlated.

The same can be said of the predicates found on the Semantic Web, as can be seen from even a small sampling of predicates operating on the DBPedia *South_End_Grounds* entity:

```
seatingCapacity
tenant
stadiumName
tenants
name
brokeGround
```

In this example we not only see predicates with values whose semantic meaning is not made machine interpretable - such as seatingCapacity and brokeGround - but also predicates with different names that are redundantly describing the same thing - such as stadiumName vs name, and tenant vs tenants.

2.3 MultiNet

Multilayered extended semantic networks (MultiNet) is both a knowledge representation and a language for semantic representation of natural language expressions. MultiNet was developed by Dr. Hermann Helbig, who used earlier semantic networks as a basis for this representation. MultiNet is

considered to be "one of the most comprehensive and thoroughly described knowledge representation systems" [12]. The MultiNet network consists of nodes and links in which nodes represent concepts and links represent relationships. The attributes of a node are broken down into three general categories:

1. **Immanent Knowledge**

   This is the knowledge that is independent of any situation or use of the concept in the description of a specific fact. There are two types of immanent knowledge:

   - *Categorical*

     This represents knowledge that is always true about the given concept. The example given in the MultiNet documentation for the concept *house* is: "A house is a building; it always has a roof and is characterized by its years of erection" [7].

   - *Prototypical*

     This represents knowledge that is usually true about the given concept. The example given in the MultiNet documentation for the concept *house* is: "The house usually has windows and (in general) an owner" [7].

2. **Situational Knowledge**

   Situational knowledge describes in what ways a concept is involved in a specific situations. This part does not affect the meaning of the concept.

This sort of separation into "always true" and "never true" is entirely dependent on the dataset being considered. The MultiNet documentation characterizes categorical knowledge as "a basic assumption which is valid as long as there is no other information available" [7]. While this sort of distinction makes sense in a global scope, if all the data in the graph is representative, it imposes stricter distinctions than seems necessary. First, if we look at a particular subscope, or subgraph, we will see that within this graph, what is considered categorical and prototypical knowledge will likely be different. For instance, if we look at houses in the context - or scope - of a particular neighborhood or street, we may find that *all* the houses have an owner. We may also find that there are additional attributes that are "categorical" in this context. For example, perhaps we are looking in the scope of a rich neighborhood in which all houses cost over 1 million dollars. Furthermore, if we were to add data to the semantic knowledge graph on war zones, we may find that houses do *not* always have roofs. Thus, we feel that the "typical" knowledge for any concept is both highly context dependent, and hardly ever absolute. As stated by proponents of prototype theory, concepts "are not characterizable in terms of necessary and sufficient conditions, but are graded (fuzzy at their boundaries) and inconsistent as to the status of their constituent members" [13]. What we think of as the meaning of a word, is not in fact an objective truth, but rather a subjective characterization that

is learned from experiences. As a result, a person's - or semantic knowledge system's - representation for any concept depends on the experiences - or dataset of experiences - that is given. The MultiNet approach to concept definition, on the other hand, is just the opposite. It is a *top-down* approach, in which the characteristics of a concept impose restrictions, or propagate down, to the instances of that concept. Prototype theory, on the other hand, dictates a *bottom-up* approach, in which a set of instances, or experiences, serves as the basis for a concept definition. The characteristics of all these individual experiences combine to define a concept.

Furthermore, although MultiNet incorporates the notion of quantity into their representation, their learning and characterization of concepts is primarily qualitative. We believe that the combination of numeric and qualitative data can provide more insight and allow more powerful learning mechanisms than qualitative data alone. Oftentimes, the creation of new qualitative descriptors and concepts has an underlying numeric basis that can be leveraged with the right algorithms and dataset. The learning algorithms we have built demonstrate this idea.

Chapter 3

The Semantic Knowledge Graph Representation

Our goal is to develop a knowledge representation that captures the meaning in natural language. Specifically, we wanted to:

- Detect semantic patterns at various levels of abstraction.
- Infer information.
- Predict future patterns.
- Learn new concepts.

3.1 Axioms

We began by outlining the basic components and operations that our knowledge representation needed to have. In doing so, we wanted to ensure that this representation was general enough to capture as many different types of concepts and semantic patterns as possible. Thus, we endeavored to define the simplest possible building blocks, which could then be used to construct any desired semantic concept or set of concepts:

1. **Nodes**

   We consider concepts, ideas, events, attributes and quantities as discrete entities, which we term *nodes*. A node, therefore, is a generic container for information of any kind and at any level of abstraction.

2. **Links**

Intuitively we know that concepts do not exist in a vacuum, but in a network of other concepts. Therefore, we introduce the idea of a relationship between nodes as a *link*. The collection of nodes and links thus form a network that is the basis for our knowledge representation.

3. **Operations**

   Clearly any knowledge representation cannot be static but must change dynamically in response to new information. Therefore, we allow our network to add, remove and redefine nodes and links.

Combining these components, the semantic knowledge *graph* G is, therefore, the set of nodes N and links L. Learning algorithms analyze G and apply operations to modify G based on the results of their analysis.

3.2 Graph Structure

3.2.1 Nodes

Every entity in our semantic model is represented as a *node* in the semantic knowledge graph, which is connected to other nodes via different *links*. A node can represent anything, from a concept, to an instance, to a particular word, to a sequence of other nodes, etc.

Concepts, Instances and Forms Generally speaking, every node falls into one of three categories:

1. **Concept**

   A concept represents some general idea or template.

2. **Instance**

   An instance represents a particular occurrence, or existing example, of some concept. This concept is referred to as the *parent* concept of the given instance. Of course, this does not necessarily mean that every instance node will have an associated parent concept - it is possible that the parent concept is unknown and thus is not linked to the particular instance in the semantic knowledge graph. It is also possible - and in fact likely - that a given instance may be an instantiation of multiple concepts. For example, the node *Andre Agassi* would be both an instance of a *tennis player* and a *man*. Both of these concepts would be considered a parent of the *Andre Agassi* node.

3. **Form**

The form is the textual representation of any given concept or instance. Thus, whenever a particular concept or instance is found in a piece of text, there is a particular word, or sequence of words that it corresponds to.

3.2.2 Links

We define three main types of links in our knowledge representation:

1. **Type**

   The **Type** link captures the conceptual "is a" relationship. A *cat* is a *feline* which is an *animal.*

2. **Instance**

   When a node A is an instance of a node B, we say that B is the *parent* of A, or A is the *child* of B. A child can have any number of parents and a parent can have any number of children. A child is connected to each of its parents by the *Instance* link.

3. **Attribute**

   Any concept or instance node may have characteristics and components; these are called the *attributes* of a node. They are connected to the node by the *Attribute* link.

Although the links in the graph are not strictly speaking one-directional, there is an implied "downward" direction that corresponds to moving from a more general scope to a more specific one. Specifically, anytime that one traverses a **Type** link in the "downward" direction, one is traveling from a more general category to a more specific one. In other words, moving in the direction *Tornado* is a **Type** of *Natural Disaster* is a move "up" the knowledge hierarchy, whereas moving from *Natural Disaster* to *Tornado* is a move "down". Similarly, given a node with some set of attributes, moving from a node to its attributes narrows the scope and is thus a "downward" move. For instance, shifting from a *Tornado* to its *Wind Speed* is a narrowing of focus and therefore a move "down". Perhaps the most obvious and useful use case is a move "down" the **Instance** link, such as a move from the concept *Tornado* to a specific instance of a *Tornado.* These kinds of shift is especially relevant when we are looking for patterns at different levels of abstraction or within different scopes.

3.2.3 Sequence and Set Representation

In describing the components of any concept, it is often helpful to alternate between thinking of a concept as:

- An unordered set of attributes or concepts.
- An ordered sequence of attributes or concepts.

This gives rise to two types of representations: sequence and set representation.

Set Representation

Set representation is unordered. A *set* of nodes S represents a group of nodes $N_0, N_1 ... N_m$ that are all connected to some node P by the same type of link. Thus, we will often refer to the *set of attributes* of some node P, which means all nodes connected to node P by an **Attribute** link. Similarly, the *set of instances of type* T refers to all instances that are connected to some node T by an **Instance** link. Finally, the *set of concepts of type* T refers to all nodes connected to some node T by the *Type* link. In later sections we may refer to sets of instances as *clusters*. A *cluster* of type T is the set of all instance nodes connected to the node T by **Instance** links. Sets can be partially overlapping. For example, the set of instances of type *actor* would overlap with the set of instances of type *director*, because there are people who are both actors and directors. However, since an actor is not a type of director, or vice versa, neither subset would be entirely contained in the other.

Set representation is useful when we wish to learn order-independent patterns about a group of nodes. For instance, if we wish to learn the set of attributes that is common to all dogs, we do not care what order they are listed in, just that this set is as specific as possible. We would learn this kind of pattern by examining the set of all instances of the "dog" concept. It is important to note here that a set can span multiple levels or a single one, depending on the task being performed. Thus, if we are looking at the set of all dogs, we would keep moving down the hierarchy through levels such as "big dog", "golden retriever", etc. As long as an instance falls anywhere in the "dog" hierarchy below the "dog" concept node, is is considered to be part of the set of all dog instances. On the other hand, when we are talking about a specific dog instance and we wish to obtain the set of attributes of this dog, we do not keep following down the hierarchy of **Attribute** links. For example, if we have an instance of *my dog Sparky*, which has an attribute of *nose* and that *nose* has an attribute of *cold*, then *cold* is *not* in the set of attributes of *my dog Sparky*.

Sequence Representation

Sequence representation is order dependent. A *sequence* of nodes may represent the sequence of words in a sentence, or a sequence of events, or any other group of nodes that must be moved through in a particular order.

Sequence representation is particularly useful for prediction because we can infer what the "next" node in a given sequence should be based on past patterns. Thus, if we repeatedly see sequences of events such as "Lucy procrastinates on studying." $\Rightarrow$ "Lucy gets a D.", "Jack puts off studying." $\Rightarrow$ "Jack gets an F.", we would eventually learn the pattern "Student delays in studying." $\Rightarrow$ "Student gets bad grade.". Once we learned this sequential pattern, we would then be able to apply it to all situations in which a student delays studying to predict that the student will likely get a bad grade. This type of sequence would be referred to as a *temporal* sequence, because the nodes are ordered

based on time. Another common type of sequence is a *spatial* sequence, in which nodes are ordered based on some spatial dimension. An example of a spatial sequence could be something like: *a book* is under *the pile of clothes* which is under *the desk*. Treating a sentence as a sequence of words, or more precisely, concepts, is another common type of sequence. Thus, if we see a large set of sentences similar to "Jane throws the frisbee.", "Jack tosses the baseball.", and then see the beginning of a sentence "Joe threw ..." we would guess that Joe will throw some sort of sport projectile. We would make this prediction on the basis of a higher-level pattern *name word* $\Rightarrow$ *throwing motion* $\Rightarrow$ *sport projectile*.

3.2.4 Numeric Nodes

Certain nodes in our semantic knowledge graph represent ideas that are quantifiable. Examples of these types of nodes include everything from the value of an x-coordinate, the temperature, the time, a quantity, etc. The existence of these numeric nodes adds an extra dimension to the pattern-matching process, making it possible to look for higher-level numeric patterns, as well as purely conceptual ones.

Numeric nodes allow for the establishment of *greater than* and *less than* relationships between nodes. These *greater than, less than* relationship should exist within a clearly defined scope - a numeric temperature node should be compared to only other temperature instances, a numeric time node should be compared only to other times, etc.

In fact, upon closer examination it becomes apparent that the existence of the *sequence* as described in the previous section is contingent upon this very concept. Essentially, we are imposing an order - a *less than* relationship - on all the nodes that are in the local scope of that ordered node. Thus, a node that captures the sequence *The cat sat on the mat* is just defining a *less than* relationship of *The* $<$ *cat* $<$ *sat* $<$ *on* $<$ *the* $<$ *mat*.

Applying this idea to an example with numeric nodes, if our graph contains many different instances of *temperature* nodes, we can imagine that all these numeric nodes are simply parts of the sequence of an ordered super-node – the *temperature* sequence node. In this case, the scope of this ordered node is the *temperature* type (as opposed to the local *sentence* scope of the previous example).

This kind of definition makes it possible to look for numeric patterns and connect them to existing relations or concepts. To give an example, let's say there is a node which describes the relation between two other nodes, such as: *The lamp is above my head.* where *above* is the relation that ties together the nodes *lamp* and *my head*. Let's further suppose that the nodes *lamp* and *my head* include attributes that describe their locations - specifically, x y and z coordinates. A numeric analysis of all the instances of type *on top of* should reveal that there is a higher-level numeric pattern among all these nodes: *y-coordinate of first object* $>$ *y-coordinate of second object*.

Therefore, the relationship of *above* would now have an identifiable numeric component that could be applied to any other nodes with y-coordinates. This discovery would then allow us to generate *above* nodes for *any* two objects whose y-coordinates matched the required relationship. Ultimately, this would allow our program to generate and communicate semantic discoveries in a textual, user-meaningful way - it could tell the user "The hat is above the floor" without ever being explicitly given that sentence.

3.2.5 Negation

In deciding what properties our representation should have, we considered adding the ability to have "negative" links. These would be used in situations when we learned patterns such as: (1) a node N will definitely *not* have an attribute A, represented by a negative attribute link (2) an instance N is definitely *not* the child of concept C, represented by a negative instance link. By contrast, all of the links described in previous sections represent positive, or existing, links. One of the difficulties in adding negative links to the representation, is that it is difficult to discover universal negative properties. Since the typical attributes of any given concept vary greatly between contexts, what may seem like a negative property in some contexts may not hold in others. For example, let's say we deemed that the "horse" concept will never have an attribute "wings". The moment we got to a text on Greek myths, or any other work that contained a reference to Pegasus, our negative attribute link would no longer hold. Thus, while it makes sense to assert that, overall, horses typically do not have wings, it is much harder to uphold a negative assertion that horses *never* have wings, except perhaps in very specific contexts. Even if a text explicitly states a negative relationship such as "Mary had never dyed her hair." This relationship may change if, later in this same text, we discover that Mary changed her mind and dyed her hair blue. Thus, this negative attribute would hold within the very specific context of "the first fourth of the novel about Mary".

On the other hand, we are much more likely to discover less absolute negative relationships, such as negative correlation. A negative correlation represents a relationship such as: "presence of attribute A inhibits presence of attribute B". For instance, if we see that a tornado instance is described as "weak", we are less likely to see attributes such as "severe destruction" or "long lifetime". Thus, the presence of one attribute in a node makes it unlikely to see others in the same one. If, within some context, we *never* see two attributes occur in the same node, while individually they each occur frequently, we can say that these two attributes are *mutually exclusive* within some context. Thus, if we never see tornado instances called both *weak* and *violent*, we say that these attributes are mutually exclusive for tornadoes.

3.3 Graph Properties

3.3.1 Context

As mentioned previously, by moving through the knowledge graph in the "downward" direction, we can narrow our search space to any desired scope or context. Conceptually, we can see how this ability is useful and relevant to our daily thinking - we make inferences based on our ability to both (1) consider overall trends and (2) consider situations most similar to the current one. For instance, if a person is considering the risks of a medical procedure, he or she is likely to start by looking at overall success rates for this procedure and then narrow their analysis to smaller scopes such as:

- The subset of all such procedures where the patient is in a certain age group.
- The subset of procedures performed by a specific surgeon.
- The subset of procedures performed within the last five years.

All of these examples represent different scopes that help subdivide the set of all instances into subgroups. Within each subgroup there are attributes that are unique or more common to that group and make the task of trend discovery and prediction more likely to give accurate results.

One of the advantages of our representation is that we can choose any desired instance node N in the graph and use it to define the scope, or context. If we traverse the graph in the downward direction from node N, any node we touch becomes part of the subgraph that represents the "scope of node N". For example, if we choose an instance node that is a particular article, the subgraph of this node would contain all information that is found in the article, and would thus represent this article's scope. Any node inside this subgraph would have a context of N. Since contexts can be nested, a node may belong to an arbitrary number of contexts.

One of the improvements that we have considered making is to make this concept of context more explicit by allowing a node to point to an entire subgraph. This eliminates the need to traverse the graph in the downward direction from the node to find the subgraph each time. This would also make it easier to look up what contexts a given node is in, since each context is clearly defined and labeled.

3.3.2 Going From Instance to Concept

One of our goals for our semantic knowledge representation was that it be able to learn and grow given some initial dataset. As such, we developed processes through which the analysis of instance nodes discovered patterns which add new nodes to the graph.

When the graph receives information, every aspect is instantiated; that is, every piece of information becomes an instance node. If the dataset is given a sentence such as "I jogged through Central

Park.", it will convert it into a graph representation in which there is an instance node for each of the following: (1) the entity referred to as "I" e.g. the narrator within the current context (2) the act of jogging (3) the spatial preposition "through" (4) the location referred to as "Central Park". Even though the concept of "jogging", the preposition "through" and possibly even the location "Central Park", would likely already exist as concepts in the representation, we need to create the specific instances of these concepts that occurred in the current context. This is especially important since a concept's attributes may vary by context. For example, if we were reading data from children's books, we would likely learn many unusual and unrealistic facts such as: dogs can be bright red and as big as a house, unicorns exist, animals can talk, etc. In such a situation, it is important to distinguish between the concept of real-world animals and their characteristics and children's books animals and their characteristics. In fact, we would likely want to create a separate subconcept within the "animal" hierarchy that corresponds to "fictional animals", which would have markedly different typical characteristics than "real animals". Chapter 5 talks about some of the algorithms we use to accomplish these kinds of learning tasks.

In general, the process through which we create new concepts involves seeing the same pattern over an over again within some context. Once a pattern occurs often enough, we infer that this pattern likely corresponds to some concept. This concept may or may not have a name that is known to the semantic knowledge representation. Suppose we learned that the instance nodes $N_1, N_2 ... N_m$ all followed the same pattern. The representation would start off by creating a new unnamed concept C and indicating that it is the parent of all these instances. If the named concept is already present in the graph as concept A, the representation would have to learn to merge the newly created unnamed C with the existing named A. This merging process would involve identifying that A is the parent of each of the instances N_1 through N_m. This would then allow the representation to learn that the concept A has a newly discovered characteristic pattern. This acquired knowledge could then be propagated to all other known instances of A in this context. If there was no named concept A, the representation would leave C unnamed until it received further information that would contain such an A.

One of our intended areas of further work is to determine the best way to identify when a discovered concept C already exists as some other named concept A. This would require the existing instances of A to have enough similarities with the instances of C that we could make this judgment call with a high percent confidence. One of the challenges of this is that the typical characteristics of a single concept may vary widely depending on context. For example, a technical biological description of a plant would contain many attributes that would not be found in a conversational description of a plant. Similarly, if a particular text is promoting, or biased in favor of, some concept, it would use very different descriptors from a text that was biased against this same concept. If the representation is given a large enough dataset, it would be able to identify when a specific context

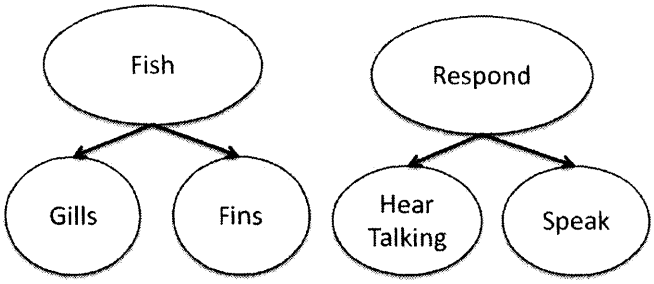

Figure 3-1: Typical Attributes for Different Concepts

describes a concept very differently from the majority of others. For example, if we used only liberal media sources to learn the characteristics of the concept of "tax cuts", the representation would notice, given a conservative media source, that these new instances of "tax cut" had very atypical characteristics. This would allow the representation to create sub-concepts corresponding to "tax cuts in the context of liberal media" and "tax cuts in the context of conservative media". The instances of tax cuts would then be subdivided and adjusted to have these new parents as concepts.

3.3.3 Concepts as Filters

Extending the idea in the previous section that every concept has a set of "typical" attributes, we can think about a concept as a filter on a set of attributes. Suppose we have learned, as described in the previous section, that the concept C has a set of typical attributes: S_C. Given some node N with a set of attributes S_N, we know that concept C is the parent of N if S_N contains S_C. As such, in order to identify whether the parent of node N is concept C, we can filter the set of attributes associated with N - S_N - on S_C.

For instance, suppose we have learned that the concept of "mammal" has typical attributes "has hair", "bears live young" and "breathes air". We could then take any animal and filter out any attributes that did not match the ones of "animal". If this filtering process produced a set that was identical or similar enough (by some definition of "similar enough") to the "animal" attribute set, we would know that the current animal is, in fact, a mammal. Since, as mentioned before, different contexts may have varying typical characteristics for a concept, we would need some measure of a "close enough" match to allow for likely variation in the data.

One of the powerful aspects of our representation is that this filtering process would be performed the same way, independent of the type of concept being considered. Figure 3-1 illustrates this idea by showing two different concepts "respond" and "fish", both of which have typical attributes that can be used as filters. Thus, the verb "respond" can be thought of as a filter on events, while the noun "fish" can be thought of as a filter on objects, or physical characteristics.

Chapter 4

Implementation

Our implementation of the semantic knowledge representation, which was written in C#, has the following structure:

1. **Input Layer**

   The application takes input either (1) in the form of text, which is then parsed and converted into the internal graph representation (2) in raw graph form, which uses special syntax to indicate nodes and links.

2. **Graph Representation**

   Within our application we have a class named *Memory* which maintains the graph data structure, and contains methods to manipulate and retrieve parts of the graph.

3. **User Interface Layer**

   Our application has a user interface that displays the graph both in text form, and in graphical form.

4. **Learning Layer**

   This layer contains various algorithms that look for patterns in the graph and suggest changes that should be made to it to "learn" new ideas and concepts.

4.1 Input Layer

The application contains a parser that reads in text and converts it into the internal graph representation. This method of input is particularly powerful because it means our system does not require a customized input - it can read content straight from websites, books, or any other text.

For the purposes of this thesis, I used the second input method which takes in raw graph form, using special syntax to designate nodes and links. Specifically, we represent concepts as follows:

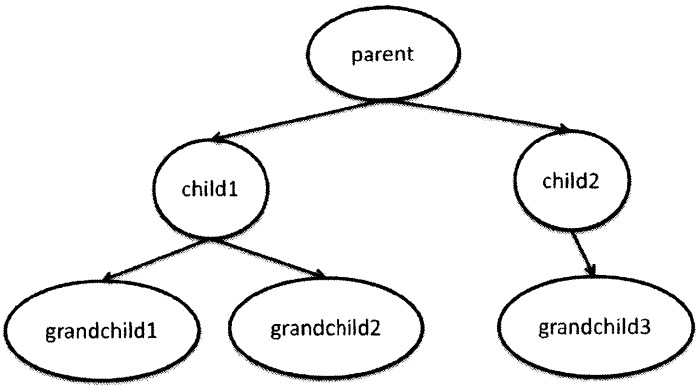

Figure 4-1: Concept Graph Generated from Input

`nameofconcept.nameofparentconcept`

We use this form for easy recognition and simple disambiguation. An example of this would be:

`bat.mammal`

We implemented XML and JSON formats, but for ease of use and simplicity of format we developed the following specification:

```
parent>
    child1>
        grandchild1
        grandchild2
        .
    child2>
        grandchild3
        .
    .
```

As shown above, a right arrow '>' at the end of the line indicates a move one level down the hierarchy, while a '.' indicates a move one level up the hierarchy. From this example, we see that the full form of *child1* is *child1.parent* and the full form of *grandchild1* is *grandchild1.child1*. This input would be transformed into the internal graph structure shown in Figure 4-1.

If we wish to designate that a particular node is an instance, we start the line with the "instance" keyword, followed by the name of the parent concept being instantiated. If no concept name is explicitly given, the instance does not have a parent concept. Similarly, if we wish to designate that a particular node is an attribute, we start the line with the "attribute" keyword. Anything that is not explicitly defined as an instance is considered a concept.

Thus, if we wanted to represent the event of a tornado crashing into a building, it would look something like this:

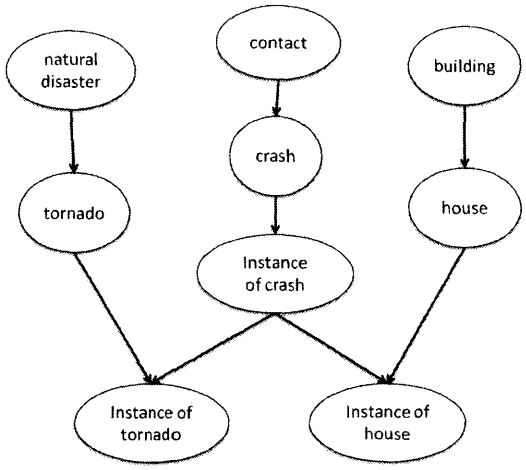

Figure 4-2: Sample of a Graph Generated Representation from Input

```
instance crash.contact>
   attribute instance tornado.natural_disaster
   attribute instance house.building
```

The above raw graph input would be turned into the graphical representation shown in Figure 4-2.

If a node is numeric, the value of the node is added at the end of the line. For instance, if the above crash event took place in a video game, we would see numeric attributes such as:

```
instance crash.contact>
   attribute instance tornado.natural_disaster
   attribute instance house.building
   attribute instance x_coordinate.coordinate 609.7529
   attribute instance y_coordinate.coordinate 76.21459
   attribute instance z_coordinate.coordinate 1075.424
```

If we wish to refer back to an earlier mentioned node, we create a unique identifier and place a "/uniqueid" at the end of each line that refers to this node. For example, if create an instance of a tornado and then referred back to it every time an event occurred involving it, we would write that as follows:

```
instance tornado.natural_disaster/27>
   attribute instance violent
.
**** Some other input text ****
instance tornado.natural_disaster/27>
   attribute instance destroy.damage>
      attribute instance house.building
      .
   .
```

As shown, the unique identifier 27 is used to refer back to the tornado instance created in the first line.

4.2 Graph Representation

The data structures used to store the graph, along with the methods used to manipulate the graph, are encapsulated in a class called Memory. The Memory class assigns unique IDs to all nodes and links. A node is represented using a Node class, which stores the name, ID, type and numeric value of the node. Of these four, only the ID and type are required, since a node may not be numeric or may not have a name. The type of the node is either **Concept**, **Instance** or **Form**. Links are represented using a Link class which stores the ID of the link, the IDs of the (1) start or "from" node (2) end or "to" node, and the type. A link can have type **Type**, **Instance** or **Attribute**. All of the above fields are mandatory for a link. Some of the common operations that are performed on a graph are:

- Add or remove a given node or link to or from the graph.
- Retrieve the parents of a given node.
- Retrieve the children of a given node.
- Retrieve the set of attributes of a given node. For this action, there is also the option of retrieving the attributes grouped by parent type.
- Check if a node A is an *ancestor* of node B. An ancestor of a node is any node in its parent hierarchy e.g. a parent of B, or a parent of a parent of B, etc.
- Retrieve all ancestors of a node.
- Find the first common ancestor of two nodes A and B. This is obtained by starting at nodes A and B and traversing up their parent hierarchies until we find the first intersection point. It is possible that the two nodes do not have any common ancestors.
- Obtain a measure of similarity between to nodes. This measure is described in more detail in the following section.

4.3 User Interface Layer

Our application provides a user interface for displaying the internal knowledge graph representation, both in text and graph form.

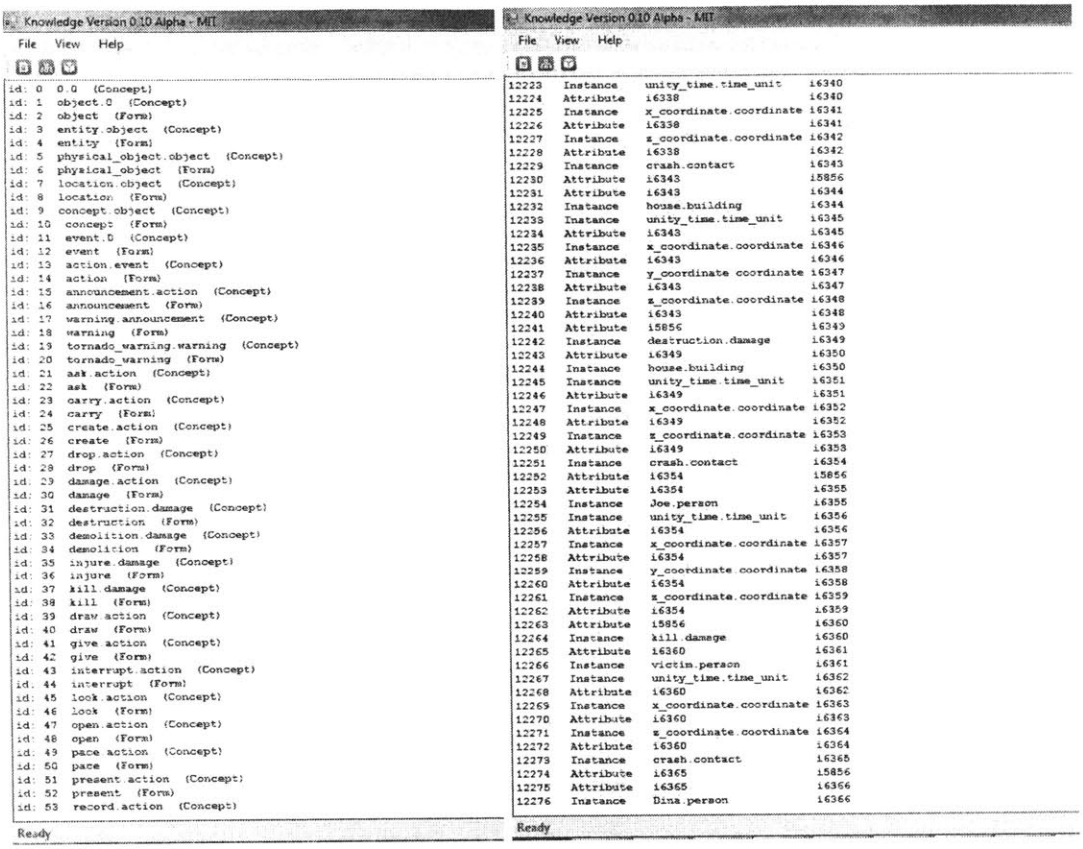

Figure 4-3: Knowledge Application Text Output

The text form output is shown in Figure 4-3. The first image shows the text form of nodes, where each line gives the ID, name and type of a node. If a node does not have a name it is given a pseudoname which consists of (1) 'i' + ID for instances and (2) 'c' + ID for concepts. All forms have a name by definition. The second image shows the text form of links. Each line gives the ID and type of the link, as well as the names of the "from" and "to" nodes.

The graph form is shown in Figure 4-4. The user can select any particular node in the graph and specify how many levels down to display connected nodes. Scrolling over any node allows the user to view all of its characteristics: name, ID, type, numeric value, as well as the parents.

4.4 Learning Layer

The application has various algorithms that it uses to detect different types of patterns in the dataset. These are described in the next chapter. The output of these algorithms takes many different forms, from descriptions of "typical" attributes for a certain concept, to discovered numeric relationships among nodes, to groupings of similar nodes. Most of these outputs fall into one of two categories:

1. Output that describes the current state of the graph.

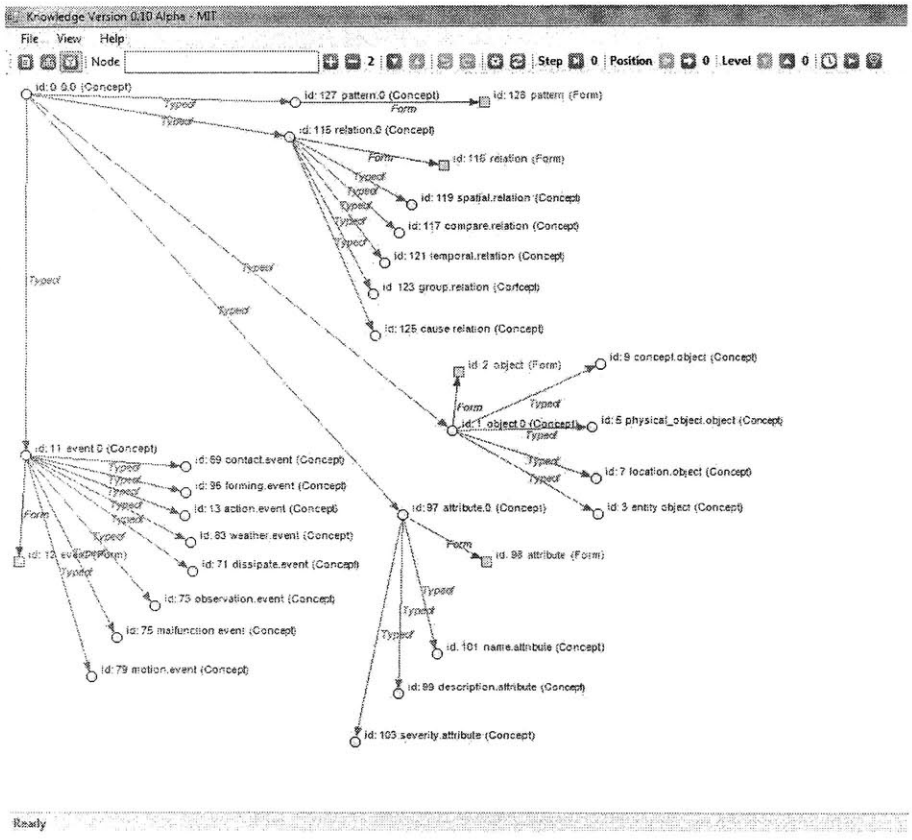

Figure 4-4: Knowledge Application Graph Output

2. Output that recommends how the graph should be modified. This type of output usually suggests new concepts that should be added to the graph.

Chapter 5

Learning

In designing our knowledge representation, we wanted to not only represent semantic relationships and entities, but also to discover the rules and patterns that these relationships and entities adhere to. This would then enable us to apply these rules to make further inferences, and reason about the dataset. Ultimately, we wanted to develop algorithms that would allow our system to *learn* the meanings of both existing - or given - concepts and new concepts.

5.1 Algorithms

Learning the meaning of a concept requires the ability to give a *definition* - or to name the attributes that instances of this concept have. Since we subscribe to the idea behind prototype theory - that any concept definition is inherently fuzzy and context-dependent - we wanted our algorithms to (1) allow for variability within a concept and (2) to provide definitions with reference to a particular scope, or context. Allowing for variability means that two instances do not have to match *exactly* to be children of the same concept, they simply have to be "close enough", using some measure of "close". To ensure that definitions are context-dependent, we employed a bottom-up approach to definition learning. A bottom-up approach means that a definition for a concept is learned by examining all instances of the concept *in a particular scope*, and then generating the definition for the concept based on that set of instances. With this in mind, we designed the following algorithms to learn the meanings of both given and discovered concepts.

5.1.1 Node Comparison

One of the abilities we wanted our semantic knowledge graph to have, is the ability to identify an unnamed, or imprecisely named, node. By *imprecisely named*, we mean that the given parent of the node is too general. For example, if we have an instance of a "coffee table", that was labeled as an

instance of "furniture", this would be an accurate but overly general label. Assuming that the graph would already have numerous instances of this type of node, it would need to perform a comparison and determine that the unnamed node most closely resembles an instance of "coffee table". In order to do this, we need some measure of similarity between two nodes.

For this purpose, we defined an *exact match weight* between nodes A and B. This match weight consists of a weighted average of (1) the similarity of the parents of A and B (2) the similarity of the attributes of nodes A and B.

Similarity of Parents If we are given two nodes A and B, we need a way to measure how similar the parents of these two nodes are. Intuitively, we know that if A and B have the same parent concept, then the match should have a weight of 1. Furthermore, if the parents of these two nodes are in the same concept hierarchy, their similarity measure should be greater than 0 but less than 1. For instance, using the previous example of the "furniture" and "coffee table" instances, we would want to assign these two parents a non-zero match weight since a coffee table is a type of furniture. The closer the two parents are to each other in the concept hierarchy, the higher the match weight. Suppose that the concept hierarchy looked like this: *furniture* $\Rightarrow$ *table* $\Rightarrow$ *coffee table*. We would define the parent match weight between the concepts "furniture" and "coffee table" as a function of the distance between them - which in this case is 2 levels.

Intuitively, we know that each time we go up a level in abstraction, we lose some portion of meaning, or information. When we move from the concept of "coffee table" to "table" we lose the specific type of the table. When we move from "table" to "furniture" we lose the specific type of furniture. Thus, the further away two concepts are from each other in a hierarchy, the less precise the parent match. This suggests that there is some *decay factor* F that applies for every level that is added to the distance. Thus, we use the following formula to calculate the weight of a parent match, given two parents P_A and P_B that are in the same hierarchy: $W = F^{|P_A - P_B|}$, where $|P_A - P_B|$ is the distance, or number of levels between the two parents. Thus, if we use a decay factor $F = 0.5$, we obtain that the weight of the parent match between "coffee table" and "furniture" is $0.5^2 = 0.25$.

Since both nodes A and B can have multiple parents, we need to split the parent similarity among the individual parent match weights. Suppose node A has m parents and node B has n parents, each match of a parent of A to a parent of B would have a maximum weight of $\frac{1}{Max(m,n)}$. Thus, A and B cannot be an exact match unless they have *all* of the same parents. The parent similarity would then be the combination of individual parent match weights that gives the highest possible value.

If either of nodes A or B does not have *any* parents, then this node is a completely unknown entity and thus, no parent similarity can be calculated. In this case, the parent similarity is not included in the similarity calculation.

Similarity of Attributes In order to find the similarity of the attributes of two nodes A and B, we need to match attributes of A to attributes of B in a way that optimizes the total match weight. Thus, for each possible pairing of attribute of A to attribute of B, we calculate the attribute similarity and then take the combination of pairings that has the highest total weight. Each attribute pairing comprises a portion of the total attribute similarity match weight. This portion is equal to $\frac{1}{Max(N_A, N_B)}$ where N_A and N_B are the number of attributes of nodes A and B.

To calculate the similarity of two attributes T_A and T_B, we need to recursively calculate the node match weight of these two attribute nodes. In the base case when T_A and T_B have no attributes, the node match weight becomes just the parent similarity weight.

Obtaining the Match Weight To combine the parent similarity with the attribute similarity, we calculate the weighted average of the two. In order to do so, we need to pick what portion of the total match weight should be the parent similarity and what portion should be the attribute similarity. Since the parents are located one level away from the two nodes being matched, it makes sense to assign the parent match a portion F of the total weight, where F is the decay factor. Thus, the final match weight is $M = F \cdot W_P + (1 - F) \cdot W_A$, where W_P is the parent similarity and W_A is the attribute similarity.

Choosing Reasonable Parameters In order for this node matching calculation to produce meaningful match weights, we needed to choose a reasonable value for the decay factor F. Since the decay factor represents the proportion of information lost for each move up a level, it should be a function of the total number of levels in the graph. This idea stems from the intuition that the highest node in the knowledge graph contains the least possible amount of detail, while the lowest node in the knowledge graph contains the most possible amount of detail. If the graph contains many levels of concepts, then each move up a level loses a correspondingly smaller fraction of meaning. The more levels are added, the higher the level of specificity, or detail. In Chapter 7 we compare the performance of our algorithms using several different values for the decay factor. By doing so, we can empirically obtain the decay factor that gives the most accurate results.

Furthermore, it is important to note that although this node matching algorithm is an exhaustive and intuitive approach to calculate node similarity, in practice, it is too computationally expensive to scale for larger graphs. Thus, later in this section we explore a more efficient way to determine the likely parent for an unknown node. In doing so, we use many of the same components and parameters as the node matching algorithm.

5.1.2 SuperNodes

As described in Chapter 3, numeric values of nodes can be thought of as indexes that allow numeric nodes to be placed relative to one another in a sequence. This becomes a very useful way of thinking about numeric values because it allows us to take all numeric nodes of one type, place them in a sequence and look for patterns across these numeric values. Depending on the subgraph, or context, being considered this sequence of numeric values may contain different types of patterns.

For instance, given an article about the fitness of different athletes, if we were to take all instances of Body Mass Index (BMI) quantities and place them in a sequence, we would then be able to observe various properties of BMIs of athletes. We could observe the range of BMI values, the frequencies of certain values, the average BMI for an athlete, etc. If we were to then consider a BMI sequence in a different context, perhaps BMI values for American adults or teen athletes, we would likely discover different sets of values and patterns within those sequences.

Since each numeric value is a component of a certain *type* of node, it only has meaning within the context of that type. Thus, the concept of "height of 6 ft" is an entirely different one from the concept of "weight of 110 lbs", and it would not make sense to attempt to place these two different numbers - 6 and 110 - into any sort of sequence. On the other hand, given a set of height values "6 ft", "5 ft" and "5.5 ft", it would be reasonable to establish a sequence of type "ft", in which the numeric values 5, 5.5, and 6 would be placed in increasing order.

It is important to note that the "type" of a numeric node, is not simply the unit of the numeric value - it also defines the context of the value. For example, although we could undoubtedly define a numeric type that is simply "inches" this would likely lead to the creation of a fairly useless sequence of values given a heterogeneous enough dataset. It would hardly make sense to put, say, the radiuses of lampshades (in inches) in the same sequence as widths of computer monitors (also in inches). Thus the type of a numeric node should define a specific concept, such as width of lampshade, along with the specific unit being used. Once we have narrowed down the type of the numeric node to the desired scope, we could then learn and discover meaningful patterns about the defined sequence of values.

To enable this sort of type-specific sequence value analysis, we created the concept of a "supernode". A "supernode" is an ordered node which is a sequence of all numeric values of one type within the graph. When a graph is first created, we find all the different numeric node types and create a supernode for each of them. Thus if we were given a dataset about tornadoes, we would likely end up creating supernodes such as: "average wind speed in MPH", "average wind speed in KM/Hour", "'trail length in ft", "lifetime in minutes", etc.

The pseudocode for this supernode creation process is shown below:

```
typeToSuperNodeMap = {}
```

```
// maps type of numeric node to supernode for this type

typeToNumericNodesMap = {}
// maps type of node to list of all numeric nodes of that type

types = {}
for each node in the graph:
   if node N has numeric value:
      parentNodes = parents of N
      for each parentNode P in parentNodes:
         if types does not contain P:
            types <== add key pair (type, empty list)
         add N to types[P] list

for each type P in types:
L = types[P] list
   CreateSuperNode(P, L)

function CreateSuperNode(type P, list L):
   Sort L
   Create a new ordered node S
   // S = supernode

   for each node N in L:
      Create attribute link from S to N
   typeToSuperNodeMap <== add keypair (type P, S)
```

5.1.3 Relational Numeric Algorithm

In examining common words, phrases and texts, we observed that the meaning of words or concepts is often an abstraction of an underlying numeric relationship. Any word that establishes the state of one thing *relative* to another is using some - possibly subjective - measure to define a relationship.

Some of the most obvious examples of this are prepositions: specifically, ones that convey relative positions of entities, both in time and space. For instance, prepositions such as "behind", "above", "below", represent spatial relationships that could be easily discovered by examining and comparing relative *positions* of the described entities. Similarly, prepositions such as "before" and "after" represent temporal relationships that could also be easily discovered by comparing relative *times* of the described entities. We can think of these prepositions as establishing a mathematical "greater than" or "less than" relationship among two entities, using some chosen numeric frame of reference. Thus, the concept of "above" must be defined within a frame of reference that defines "up" and "down" along some numeric distance axis. For examples, If we were to define "above" within a computer simulation, we would likely represent it as the idea that the $y\text{-}coordinate(Object1) > y\text{-}coordinate(Object2)$. In other words, we would choose the y-axis as the numeric frame of reference.

Although prepositions provide the most obvious illustration of this idea, there are countless other examples of concepts that enforce a numeric relationship on their attributes. Verbs such as "speeding

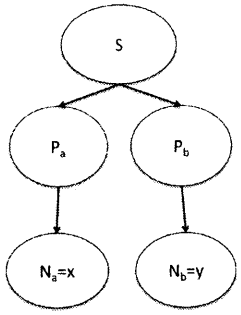

Figure 5-1: A Node that Satisfies the Greater Than Relationship

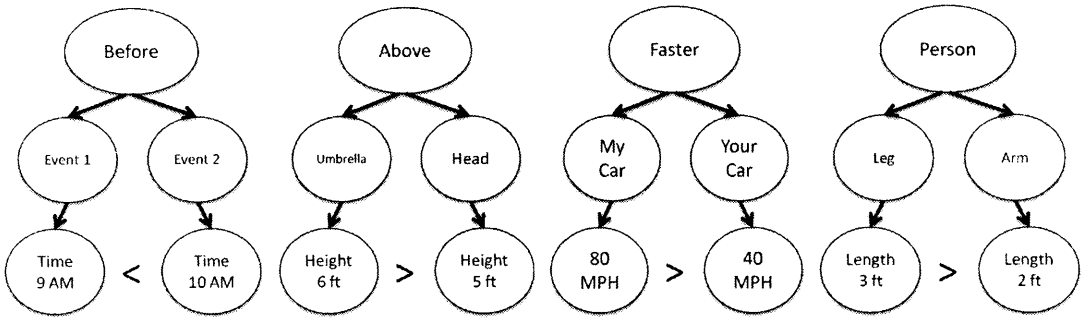

Figure 5-2: Different Types of Concepts that Satisfy the Greater Than Relationship

up", "prolonging", "growing" and many others represent an increase or change that is quantitative in nature. Whether the actual quantity is speed, length, size or any other measurable entity, the numeric relationship captured is the same: a "greater than" or "less than" relationship. This numeric pattern manifests itself not only in prepositions and verbs but throughout the language: "harder", "more", etc.

Furthermore, any attributes of an entity whose relative measures stay constant follow the same pattern: a person's legs are longer than his arms, the wind of a tornado has a much wider radius than its funnel, the height of a wall will exceeds its thickness. All of these ideas would be colloquially referred to as "common sense", but are in fact intuitions, that we gain from repeated exposure to the same pattern. Identically, we want our semantic knowledge model to acquire its own intuitions or "common sense" notions from observing reoccurring patterns in semantic data.

Thus, in order to make this kind of learning possible, we developed an algorithm that would identify consistently recurring "greater than" or "less than" relationships in numeric attributes. This relationship is shown in Figure 5-1. It is important to note that all the above-mentioned examples are captured by this relationship, because they are represented identically by our semantic knowledge model, as illustrated in Figure 5-2. This highlights and reinforces one of the strength of our knowledge representation - it allows for many conceptually different types of patterns to be discovered using the same strategy for learning.

The pseudocode for this numeric algorithm is shown below:

```
for each numeric dimension D:
   G = {} // set of all "greater than" pairs
   pairOfAttributesToNodeMap = {} // maps each "greater than" pair in G to the
                                  // node that has this pair of attributes
   N = set of all numeric nodes in dimension D

   for each pair of nodes (Na, Nb) in N:
      Pa = Parent(Na)
      Pb = Parent(Nb)
      check if Pa and Pb are both attributes of any node S

      for each such S:
         if Na > Nb:
            add pair (Pa, Pb) to G
            add key = (Pa, Pb), value=S to pairOfAttributesToNodeMap
         else if Nb > Na:
            add pair (Pb, Pa) to G
            add key = (Pb, Pa), value = S to pairOfAttributesToNodeMap
```

5.1.4 Range Numeric Algorithm

Numeric ranges represent another type of quantitative pattern that often serves as the basis for concepts. Words such as "strong" or "medium" or "short" are all abstractions of numeric ranges, whose parameters we learn from daily life experiences. These parameters are very much context-dependent, so over time we develop numerous different sets of ranges, each applicable to different scenarios. Thus, we learn that while a height of 5 feet 3 inches is "short" for an American man, it is "medium" or "average" for an American woman. If we then changed contexts to Scandinavian women, for instance, we would now be in a different numeric context and would perhaps conclude that 5 feet 3 inches is "short" for a woman as well. Such contexts can be arbitrarily general or specific: we may be talking about average height for a human being, or the average weight of a newborn boy in a particular hospital. Within each context, words take on different meanings and thus different underlying numeric representations.

As such, if we were to take all numeric values of one type within some given specific context, we would be able to discover correlations between these values and the qualitative words or concepts used to describe them. For example, if we were given fifty events, each with an associated time, where each event was described using either the words "early" or "late", we could easily discover the correlation between the qualitative word attribute and the quantitative time attribute. Furthermore, even if we were not given these qualitative descriptor words, we would still be able to observe a numeric pattern - that one group, or cluster, of events all occurred around a time toward the beginning of the spectrum and another group around a time toward the other end. Thus, even though we would not know that these clusters of events should in fact be labeled as "early" or "late", we would still

have learned the *concepts* of "early" and "late", even though we didn't know what to call them. And in fact, it seems reasonable to suppose that we create words or concepts to fill just such a need - to label or name some pattern that we see repeatedly and wish to be able to refer to.

With this idea in mind, we leveraged a common numeric clustering algorithm to discover such groups of values within any given context.

DBScan: Density-Based Spatial Clustering of Applications with Noise

The Density-Based Spatial Clustering of Applications with Noise algorithm (DBScan) [4] is one of the most commonly used clustering algorithms. It was first presented in 1996 by Martin Ester, Hans-Peter Kriegel, Jorg S and Xiaowei Xu as an efficient way to discover clusters in large spatial databases with noise. These two named strengths of the DBScan algorithm: (1) the ability to scale well to large datasets and (2) to be resilient to noisy numeric data, were particularly important to us. Since any real-world dataset would not be as clean and easy to cluster as the "early" and "late" example previously described, and would likely contain much unusable or irregular data, we chose an algorithm that would be able to parse through large amounts of data points and discover possibly obscured patterns.

Specifically, we wanted the algorithm to determine likely numeric clusters by examining a set of values and identifying the regions in which these values were particularly densely packed. For instance, if we were given the arrival times for all students entering a classroom, we would expect to see high density clusters of arrival times around the start of class, such as 8:45-9:15 AM for a 9 AM class. If we were to increase the level of granularity, we would perhaps even see two clusters: 8:45-8:55 AM for earlycomers and 9:05-9:15 for latecomers.

Furthermore, we would expect this level of granularity to be a function of the context or, equivalently, some chosen parameter of "closeness" of values. Thus, if we chose to give the algorithm all numeric values within the context of arrival times for 9 AM morning classes, we would want it to have a higher granularity than if we were to give it the context of arrival times for classes throughout the day. Alternatively, we could represent this as defining two arrival times to be "close" if they are within 30 seconds of each other in the first case, or within 15 minutes of each other in the second case.

This is exactly the behavior of the DBScan algorithm, which accepts two parameters: ϵ - which is a measure of "closeness" and the minimum required number of points to make up a cluster.

The DBScan algorithm uses two basic rules:

1. If a point A is close to point B and point A is close to a sufficient number of other points, points A and B belong to a cluster C.

2. Any point X that is sufficiently close to any point in an existing cluster C, is part of cluster C.

The DBScan algorithm starts at a particular point and finds all points that are "close" to it - in other words, all points within the ϵ-neighborhood of the starting point. All of the points in this neighborhood are then added to the current cluster, if there is a sufficient amount. It then proceeds to grow this cluster by repeating this process for all points it finds in this ϵ-neighborhood. It continues to apply this step until it cannot grow the cluster any further. The algorithm then chooses a new starting point outside the cluster and repeats the process. The algorithm finishes running when it has touched every point in the dataset. Below is the pseudocode for DBScan [11]:

```
DBScan(D, eps, MinPts)
   C = 0
   for each unvisited point P in dataset D
      mark P as visited
      N = getNeighbors (P, eps)
      if sizeof(N) < MinPts
         mark P as NOISE
      else
         C = next cluster
         expandCluster(P, N, C, eps, MinPts)

expandCluster(P, N, C, eps, MinPts)
   add P to cluster C
   for each point P' in N
      if P' is not visited
         mark P' as visited
         N' = getNeighbors(P', eps)
         if sizeof(N') >= MinPts
            N = N joined with N'
      if P' is not yet member of any cluster
         add P' to cluster C
```

Assuming that the step of finding all points in an ϵ-neighborhood is implemented efficiently i.e. in $O(\log n)$ time, the runtime of the DBScan algorithm is $O(n \cdot \log n)$.

It is important to note that the DBScan algorithm can technically be applied to a dataset of arbitrary dimension, provided that an appropriate way to calculate the "distance" between two points is provided. One of the most commonly used distance measures for this algorithm is the Euclidean distance measure, which states that the distance between two n-dimensional points $A = [a_1, a_2, ..., a_n]$ and $B = [b_1, b_2, ..., b_n]$ is $\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2}$. Even though this metric can be applied to any number of dimensions, it has been shown to perform better on low-dimensional data in practice, since it becomes harder to pick a good meaningful value for a "close" (ϵ) distance as n increases. For the purposes of our implementation, the use of this metric was sufficient, since we applied the algorithm to one dimension at a time. In the future, we intend to expand this further by applying this algorithm to multiple dimensions that have been found to be highly correlated (see later section on Sub-Clustering).

Applying DBScan to the Knowledge Dataset

In order to apply DBScan to our semantic knowledge dataset, we first subdivided the numeric data by value type and then applied the DBScan algorithm individually to each numeric type. The first part of subdividing the numeric data by type is done as described in the previous **Creating SuperNodes** section. One supernode is created per numeric type. Then the DBScan algorithm is applied to each supernode to discover clusters for each numeric dimension.

5.1.5 Node Type Clustering

In designing our semantic knowledge representation, we wanted a system that would be able to grow and learn as new data was added to it. Specifically, we felt it was particularly important for the system to be able to identify an unknown entity, if it has seen similar ones in the past. Thus, if the system was given a sufficiently large dataset of natural disaster information, it should be able to identify whether a new unlabeled entity is a weak, strong or violent tornado, based on its characteristics. Furthermore, the system should be able to describe what characteristics are typical for any type of entity. For instance, all tornadoes likely have winds with a high angular velocity, a path length, a touchdown point, etc. This capability would also allow the algorithm to identify which entity instances are most unusual or most typical for each node type.

Since a node can have an arbitrarily large number of types, we wanted an algorithm that would be able to assign a single node to multiple types. Thus, if a person was both a student and a teaching assistant, we would want an algorithm that would identify this person as having both of those two nodes types. Therefore, we wanted a fairly simple clustering algorithm that would split the semantic knowledge graph into clusters by node type, in which a single node could belong to any number of clusters. This requirement meant that we could not use basic hierarchical or conceptual clustering algorithms such as COBWEB [5], which assign each node to a single cluster. As a result, we chose the Fuzzy C-Means Clustering Algorithm [3], which allows a point to belong to multiple clusters.

Fuzzy C-Means Clustering Algorithm

The Fuzzy C-Means Clustering Algorithm was first proposed by J. C. Dunn in the paper "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters" in 1973 [3]. The algorithm outputs a matrix U_{ij} where each row i represents a cluster and each column j represents a point. Each cell entry u_{ij} is a value from 0 to 1 which represents how closely point j matches cluster i, also known as the *degree of membership.*

Each point is represented as a vector of values between 0 and 1, where each value indicates whether the point has a particular attribute. For instance, if the set of all possible attributes for a natural disaster dataset were (1)"has wind", (2)"causes flooding" and (3)"creates ashes", then a

hurricane would have a vector $[1, 1, 0]$ to indicate that it had wind ($= 1$), causes flooding ($= 1$) and does not create ashes ($= 0$). Given this dataset, if we then took all hurricane points and averaged the vectors, we would obtain the *centroid* for the hurricane cluster. The centroid for a cluster is a representation of the average, or prototypical, point in that cluster. So, if we had witnessed three hurricanes, of which two caused flooding and all had winds, we would average the vectors $[1, 1, 0]$, $[1, 1, 0]$ and $[1, 0, 0]$, to obtain the centroid $[1, \frac{2}{3}, 0]$ for the hurricane cluster. Thus, we would say that the prototypical hurricane will definitely have wind, is 66.67% likely to cause flooding and will not create ashes.

In addition to an attribute having value 1 if it is present, or 0 if it is not, we assigned partial weights to attributes that are parents of present attributes. For example, suppose we added the following concepts to the hierarchy: *changed water level* $\Rightarrow$ *increased water level* $\Rightarrow$ *flooding*, where the links are **Type** links. Then the attributes "changed water level" and "increased water level" would receive weights $0 < w < 1$ for all hurricane instances, since they all caused flooding. As mentioned in the previous section, for each level we move up the concept hierarchy, the match weight for each successive level decreases. The factor by which the weight decreases is called the *decay factor*. Given a decay factor of 0.5, we would assign the attribute "increased water level" an attribute weight of 0.5, and the attribute "changed water level" an attribute weight of $0.5 \cdot 0.5 = 0.25$. This is due to the fact that the concept "increased water level" is one level up from "flooding", while "changed water level" is two levels up. Therefore, the complete attribute vectors for the three hurricane instances would be: $[1, 0.25, 0.5, 1, 0]$, $[1, 0.25, 0.5, 1, 0]$ and $[1, 0, 0, 0, 0]$ assuming the ordering of the attributes in the vectors was given as [*wind*, *changed water level*, *increased water level*, *flooding*, *ashes*].

The goal of the Fuzzy C-Means Clustering algorithm is to minimize the following *objective function*:

$$\sum_{j=1}^{N} \sum_{i=i}^{K} u_{ij}^{m} \cdot (X_j - V_i)^2 \tag{5.1}$$

where X_j is the point vector, V_i is the cluster centroid and $(X_j - V_i)$ is the distance between the two. Therefore, the algorithm tries to assign points to clusters in a way that minimizes the sum of the squares of distances from point to cluster centroid, scaled by the membership weight. Consequently, the algorithm tries to assign matrix values u_{ij} such that larger distances from point to cluster center are scaled by a smaller value, to minimize the sum. As a result, it is the shortest distances from point to cluster e.g. the points that are closest to the cluster average, that get the highest u_{ij} weights. Thus, the closer a point is the cluster prototype, the higher its degree of membership to that cluster.

The algorithm used four basic steps to minimize the objective function:

1. Choose initial V_i cluster prototypes.

2. Compute degree of membership u_{ij} values for all points based on V_i and the constraint that all u_{ij} values for a single point must add up to 1.

3. Recompute new V_i cluster averages based on membership assignments in Step 2.

4. Keep repeating Steps 2 and 3 until the cluster averages stop changing by more than some chosen δ amount.

This process enables the clustering algorithm to place all instance nodes in the semantic knowledge representation that have attributes into their type clusters. If the algorithm places a certain node N into a cluster C, it means that N has the same type as all other nodes in C. Since the semantic knowledge representation already has a built in **Type** relationship link, this allows us to calculate initial cluster prototypes using the portion of node instances whose types are already established. These cluster averages are then given as the initial V_i parameters to the C-Means algorithm.

Adding Numeric Data

In order to incorporate numeric data into the clustering process, we needed a way to *discretize* the numeric values - convert them into meaningful qualitative attributes. In general, there are several common ways of converting numeric attributes to qualitative, or categorical, ones.

1. **Binning**

   This approach separates the attribute values into intervals, or bins, and replaces each bin with the mean, median or boundaries (minimum and maximum) of the values in it. Bins can either be chosen to be *equal-width*, so that the range of values in each bin is the same, or *equal-frequency*, so that the number of values in each bin is the same [6].

2. **Cluster Analysis**

   This approach is often used to discretize numeric attributes. Given the set of all values of some numeric attribute A, a clustering algorithm is used to separate the values of A into clusters, or groups. This method produces "high-quality discretization results" because it "takes the distribution of A into consideration, as well as the closeness of data points" [6].

Since we were concerned with producing the most meaningful and useful numeric attributes, we chose the cluster analysis discretization method. Intuitively, this approach makes the most sense because it mimics the way in which we assign meaning to sets of numeric values on a day-to-day basis. When we encounter some frequently recurring set of values, we begin to label it with some qualitative word, or identifier, which comes to represent the underlying concept. For example, if we encounter values close to "3.14159" over and over again, we eventually feel the need to assign a

name to the concept that this set of values represents: π. If we continually see a group, or cluster, of arrival times after class starts, we eventually assign a name "latecomers" to the concept.

Consequently, each of the numeric clusters obtained using the DBScan algorithm described earlier was used as a numeric attribute by the Fuzzy C-Means Clustering Algorithm. For instance, if the DBScan algorithm discovered two time clusters: 10:45 AM - 10:55 AM and 11:05 AM - 11:10 AM, each of these would be considered a unique numeric attribute. If we were to then characterize each student as a vector of attributes, as described in the previous section, we would assign a value 1 for the attribute "arrival time between 10:45 AM and 10:55 AM" to all students who arrived during this time, and a value of 0 to all students who did not. Thus, the full vector of attributes contains both qualitative and quantitative attributes.

5.1.6 Sub-Clustering

In developing our semantic knowledge representation, we wanted it to learn new concepts based on the set of concepts it was given. One way in which an algorithm can do this is by learning that a given concept should be further broken down into sub-concepts. For instance, if we take a dataset consisting of tornado descriptions, we would quickly notice that tornadoes differ widely in terms of severity, the amount of damage they cause, the time they last, and many other parameters. Based on this observation, we may begin to guess that there are further concepts such as "weak tornado" and "strong tornado" that are sub-categorizations of the "tornado" concept. This is similar to the way a child may learn that within the concept of "animal" there are many more specific sub-concepts which correspond to particular animals. A "dog" has very different attributes from a "cat", which has very different attributes from a "bird". Each should exist as its own sub-concept under the general "animal" umbrella. The more detailed the dataset is, the further we can subdivide known concepts into new categories, which then themselves become concepts.

More specifically, we would learn these kinds of sub-concepts by noticing that within the set of all "animal" or "tornado" instances, certain subsets contain attributes that are distinct from others. For instance, we would notice that birds would have "wing" and "feather" attributes, while dogs and cats do not. We would notice that strong tornadoes are more likely to destroy buildings or have a higher set of wind speeds than weak tornadoes. Thus, we could learn to identify each sub-concept by the attributes that are unique, or more likely, for it.

The sub-clustering algorithm leverages this idea by examining attribute correlations within a cluster. Suppose we have a set, or cluster, C of instances of some parent concept P. This would correspond to, say, the set of all instances of animals. Within C, instances have a variety of attributes $[A_1, A_2, ...A_n]$. This would correspond to attributes such as "wings", "feathers", "fur", etc. We start by calculating the probability of any random node in C having each particular attribute: $P(A_k)$ where $k = 0, 1, 2..n$. Essentially, we ask the question: "If we take any random animal, what is the

probability that it has wings?" or "What percent of all animals have wings?" We then go through the following process for each possible pair of attributes:

```
S <== the set of all possible attributes
positiveCorrelations = {}
// set of attributes that are positively correlated

negativeCorrelations = {}
// set of attributes that are negatively correlated

for each attribute A in S:
   for each attribute B in S:
      if A != B: // don't compare an attribute to itself
         Calculate P(A|B) // the probability of A, given B
         probabilityDifference = P(A|B) - P(A)
         // how much more likely is it that a node has attribute A
         // if we know it has attribute B

         if |probabilityDifference| > THRESHOLD_AMOUNT:
            if probabilityDifference > 0:
               Add pair (B, A) to positiveCorrelations
            else:
               Add pair (B, A) to negativeCorrelations
```

The main intuition behind this algorithm is that if $P(A|B)$ (the probability of A given B) is significantly different from $P(A)$ (the probability of A), then there is a correlation between A and B. This correlation can be either positive or negative. For example, if we know that 50% of the instances in the set of "animal" instances have an attribute "wings", then $P(wings) = 0.5$. If we then go through and look for the probability that an animal has wings given that it has feathers, we would likely see that $P(wings|feathers) = 1$. Thus, the difference $P(wings|feathers) - P(wings)$ is equal to 0.5. This means that knowing that an animal has feathers increases the probability that it has wings by 50%. Similarly, if we look at the probability that an animal has fur, given that it has wings, we will see that $P(fur|wings) = 0$, whereas $P(fur) = 0.5$. Thus, knowing that an animal has wings means there is a 0% chance that it will have fur. Therefore, there is a negative correlation between the attributes "fur" and "wings". In fact, we can make an even stronger statement if we know that *no* animal instances contain both fur and wings: we would say that these two attributes are *mutually exclusive*. This is a strong indicator that these two attributes belong in different sub-clusters, since each one occurs frequently without the other. Using the same logic, we know that the attributes wings and feathers likely belong to the same sub-cluster, since knowing that a node has the second attribute significantly increases the chances that it has the first one. Thus, this process of attribute correlation allows us to create sub-concepts $C_1, C_2, ..., C_n$ in which each set of attributes would be more highly correlated than the set of attributes for C. In this example, we would end up with two new sub-concepts: C_{dog} and C_{bird}.

An important thing to note is that this algorithm is able to learn new concepts without actually knowing their names. Thus, the algorithm would not necessarily know that one of the sub-clusters it created should be called "dog" and that the other should be called "bird". But it would be able to give a description of what a dog *is* based on the attributes that are typical or unique to this concept.

Chapter 6

Datasets

Given our semantic knowledge representation and proposed algorithms for growing a knowledge graph, we wanted to choose appropriate datasets to test our hypotheses. Specifically, we proposed that a combination of qualitative and numeric data would allow us to more easily detect underlying patterns; we therefore looked for a datasets that had these characteristics.

Little World As an initial proof-of-concept, we wanted to start by building a very limited, primitive dataset which contained a small number of entities and actions. This would allow us to obtain a preliminary idea of the sorts of characteristics that our advanced dataset should have. Thus, we created a world with basic 3D geometric shapes, which moved around and collided with each other. We then generated a dataset which described this "little" world and tested our algorithms on it.

Natural Disasters For our more advanced dataset, we started by examining a common source of natural language information: news. In doing so we quickly found that many news stories involve natural disasters. At the time of this writing, these included an 8.9 magnitude earthquake and tsunami in Japan, major earthquake in New Zealand, hailstorm in China, flood in Australia and the largest tornado outbreak in history in the United States. In particular, stories about natural disasters would include: warnings about when they were likely to occur, descriptions of the damage and analyses of their after-effects. Furthermore, we observed that news about natural disasters takes on a variety of forms, from articles, to sensor data by weather services, to individual accounts on social networking sites such as Facebook, Twitter and blog posts. This provides a heterogeneous dataset with a mix of numeric and categorical attributes.

Tornadoes From April 25 to 28 a tornado outbreak of unprecedented size broke out in the Southern, Midwestern and Northeastern United States. With 336 confirmed tornadoes, it was often difficult for friends and relatives of people in the affected areas to get in contact or gain a clear

understanding of what was happening. This was partially due to the fact that tornadoes caused damage to power lines and phone towers, making many forms of communication impossible. During these times, a lot of concerned friends and family turned to social networking sites - especially Twitter - to get the most up-to-date status reports. By tracking the content, time and location of Twitter posts which described what was being observed, many were able to follow the paths of the tornadoes and receive a real-time descriptions. This kind of textual event reconstruction is a task which cannot be easily done by computers - the text descriptions in Twitter feeds are rarely standardized enough that an algorithm can parse the text and use it to reconstruct the event being described. It requires some understanding of the meaning behind qualitative descriptions. And it requires the ability to tie these qualitative descriptions to known quantitative measures. For instance, if a Twitter post said that a person just saw a tornado pick up a car, a human reader would be able to infer that the tornado is a violent one, likely an EF5 (if they were familiar with the tornado rating scale). Similarly, if a person posted a status update saying "The wind is crazy strong out there", a human reader would be able to infer an approximate range for the likely wind speed - perhaps over 100 mph. Finally, if several people posted updates indicating where they see a tornado and how it is moving - such as "It's about 300 feet north of where I am, heading west" - a human reader would be able to combine these updates and triangulate the approximate location and direction of motion of the tornado. In order for a computer program to make similar or surpassing judgments, it would need to be able to turn these qualitative descriptions into some internal semantic representation. It would then need to acquire certain intuitions about what the terms mean. Combining these two, it would then be able to make inferences and predictions.

But how would a computer program gain these sorts of intuitions? It would need to repeatedly see the co-occurrence of certain words or concepts with quantitative attributes. Given enough data, it would learn to correlate the two different forms of data and detect semantic patterns that could then be applied to similar scenarios. Ideally, this would then allow it to make both quantitative and qualitative predictions and inferences about these scenarios. With this idea in mind, we developed a dataset containing the kinds of attributes commonly used to describe tornado scenarios.

6.1 Using Simulation to Build the Dataset

Our goal was to create a dataset that was:

- Consistent enough that there were observable patterns.
- Noisy enough that it was not as regular as any standardized data input.
- Sufficiently large to permit meaningful inference.

Furthermore, we wanted the task of dataset generation to be automated so that we could vary and control the above parameters. As such, we decided to tie the dataset generation process to a modeling framework. The framework allowed models to be designed and simulated based on easily configurable input parameters. Since we wanted our model to simulate events - specifically tornadoes - for which the rules of physical reality are essential, we chose a volumetric physical model. This allowed the simulation to identify physical interactions, such as movement and collisions. As such, we tied the dataset generation process to simulations built using the Unity development tool.

6.2 Unity

Unity is a a simulation engine used to create 3D video games, architectural visualizations and real-time 3D animations. Unity can produce games that run on Windows, Mac, Wii, iPad, iPhone, Android and in browsers [15]. Development in Unity is primarily done in a version of Javascript called UnityScript, and C#. For our work in Unity, we used the UnityScript language. Unity has numerous built-in features including a physics engine, graphics engine, support for bump mapping, reflection mapping, dynamic shadows using shadow maps, and numerous others. Among its many features, Unity has the ability to model particle systems, such as water or air, which makes it capable of modeling effects such as waterfalls, rivers, floods, steam, wind, tornadoes. Finally. there are extensive online libraries, such as TurboSquid, which have pre-built models of useful video game components, such as houses, cars, people, etc. All of these features make the task of building a basic tornado simulation fairly simple and convenient.

6.3 Dataset Generation

Our dataset generation process consisted of the following steps:

1. Build a set of rules that generate entities, events and attributes in the modeled world based on probability distributions.
2. Run the simulation.
3. Construct a semantic graph based on the simulation.
4. Read the graph into our application.
5. Apply the graph analysis and learning algorithms.
6. Test the performance of the algorithms against expected values.
7. Refine algorithms and parameters based on the evaluation.

6.4 Little World

6.4.1 The Unity Little World Model

As a first step, we built a basic proof-of-concept dataset. We started by building a very simple world with a small set of entities and events. In this world, the only entities were geometric objects: cubes, cylinders, spheres and walls used to define the boundaries. These objects move around the world and occasionally collide with each other. Thus, the only possible events were collisions.

The two types of collisions in the dataset were "hit" and "crash" collisions, which had identical qualitative attributes and different only by the magnitude of the velocity. A sample "hit" collision instance is shown below:

```
instance hit.contact>
   attribute instance cube.3Dshape
   attribute instance floor.physical_object
   attribute instance x_coordinate.coordinate -7.06
   attribute instance y_coordinate.coordinate 3.96
   attribute instance z_coordinate.coordinate 8.50
   attribute instance velocity.concept 1.53
   .
```

Every "hit" instance node contains the attributes shown: the two objects involved in the collision, the location of the collision described using x, y and z coordinates, and a velocity under 3. A "crash" collision has all the same attributes, but the magnitude of its velocity is *over* 3.

In order to test entity recognition within this world, we created several unnamed entities with attributes similar to named entities in the dataset. For instance, in order to test whether an unnamed entity could be identified as a type of "hit" collision, we added an unnamed entity with the same attributes as "hit" collisions: the two colliding objects, the location and the velocity. This was done in order to test whether the node matching algorithm could correctly identify this unnamed node as an instance of a "hit" event, based on its attributes.

Furthermore, the world contained a basic qualitative descriptor: the preposition "above". An "above" relation had attributes corresponding to the two objects being related, such as "the *cube* is above the *cylinder*". Each of these objects had an associated location, described using simple x, y and z coordinates. The y-dimension in the little world corresponded to the conventional "up" and "down" orientation. A sample instance of an "above" node is shown below:

```
instance above.preposition>
   attribute instance cube.3Dshape>
      attribute instance x_coordinate.coordinate 0
      attribute instance y_coordinate.coordinate 2
      attribute instance z_coordinate.coordinate 4.95
```

```
attribute instance cylinder.3Dshape>
   attribute instance x_coordinate.coordinate -2
   attribute instance y_coordinate.coordinate -2
   attribute instance z_coordinate.coordinate -3
```

This setup allowed us to test whether the proposed relational numeric algorithm could learn the numeric pattern behind the "above" concept. Specifically, we wanted the algorithm to observe that the x and z numeric attributes were irrelevant to the meaning of the word "above", whereas the y numeric attribute contained a consistent "greater than" pattern.

6.4.2 Summary of the Little World Dataset

In all, the Little World dataset consisted of 836 nodes and 1605 links. Five concepts were parents of instances with attributes, which allowed the node matching algorithm to compare these instances to unknown entities. The two types of event clusters corresponded to collisions of type "hit" and "crash". These events had identical qualitative attributes: the two objects involved in the collision, the location of the collision (described using x, y and z coordinates) and the velocity of the collision. Quantitatively, the type of the collision was determined by the velocity: if it was less than 3, the event was labeled as a "hit", otherwise it was labeled as a "crash". The "above" relation was an instantiated descriptor of the little world, with attributes that consisted of the two objects being related. Each of these objects had attributes that identified their locations.

6.5 Tornado World

6.5.1 The Rule Set

The tornado simulation ran according to a set of starting conditions and event triggers throughout its execution. The first set of rules created an instance of a tornado and gave it characteristics that were generated using a probability distribution. This probability distribution was based on real-world tornado statistics and categorizations. Once the tornado was instantiated, it was given a set of rules for movement, which consisted of a general direction and a probabilistic change-of-direction rule. Depending on the characteristics of the tornado, different events would be triggered in the simulation. All such events, as well as the tornado properties, were written to a file in the form of semantic knowledge graph. This process was repeated fifty times to generate knowledge graph descriptions of fifty different tornadoes.

The Unity Tornado Model World

The tornado Unity scene is built to resemble a town, with houses, office buildings, street signs and people. The tornado starts off in the cloud layer and then moves to touch down at a specified location. All tornadoes touch down in the same place so that their effects can more easily be compared. From there, the tornado starts moving across the ground with an initial specified direction. At each point in time, there is some probability that the tornado will randomly switch direction.

As the tornado moves through the town, it triggers certain events depending on its severity. Any time that the tornado comes in contact with a building or person, an event is triggered. If the tornado is strong enough to pickup, carry and drop objects, it will trigger these events with some probability as well. Finally, every time that a tornado is near enough to a person, that person may make a qualitative observation, with some probability.

At the end of its pre-specified lifetime, the tornado dissipates. As it does so, it generates one last event which also provides a summary of the tornado's actions: amount of damage and destruction caused and any injuries or fatalities that occurred.

Tornado Categorization

A tornado is defined as "a violently rotating column of air, in contact with the ground, either pendant from a cumuliform cloud or underneath a cumuliform cloud, and often - but not always - visible as a funnel cloud" in the Glossary of Meteorology [1]. Tornadoes often develop from specific types of thunderstorms called supercells, which contain rotating areas a few miles up in the atmosphere. These areas are called mesocyclones. As the rainfall from the thunderstorm increases, an area of quickly descending air, or a downdraft, forms. As the downdraft gets closer to the ground, it speeds up and drags the mesocyclone toward the ground as well. To a human observer this process often looks like a rotating funnel cloud that descends from the storm cloud layer and touches down onto the ground. For a period of time, the tornado grows, using the source of warm moist air inflow until it reaches the "mature stage". Once this inflow is cut off, due to the layer of colder winds that gradually wrap around the funnel, the tornado quickly dissipates. There is great variation in tornado intensity, which is not necessarily dependent on shape, size or location of tornadoes. However, strong tornadoes are typically larger than weak tornadoes. Similarly, the path or track length of tornadoes varies, although stronger tornadoes tend to have longer track lengths. For violent tornadoes, typically only a small portion of the track has a violent intensity. The damage caused by tornadoes also varies in intensity from (1) damage to trees, but not substantial structures, to (2) ripping buildings off their foundations and deforming large skyscrapers [14].

Tornadoes are typically broken down into three categories of severity: weak, strong and violent. Each of these severities corresponds to a certain range of characteristics, as shown in Figure 6-1 [2]. Thus, the majority of tornadoes are weak tornadoes, which are the least likely to cause

	Weak	Strong	Violent
% of All Tornadoes	69 %	29 %	2 %
% of All Tornado Deaths	< 5 %	< 30 %	70 %
Lifetime	1-10+ minutes	20+ minutes	Can exceed 1 hour
Wind Speeds	< 110 mph	110 – 205 mph	> 205 mph

Figure 6-1: Characteristics of Tornadoes by Severity

EF Rating	Average Wind Speed (mph)
0	65-85
1	86-110
2	111-135
3	136-165
4	166-200
5	Over 200

Figure 6-2: Wind Speed of Tornadoes by Enhanced Fujita Rating

fatalities, have the shortest lifetimes and have the lowest average wind speeds. Strong tornadoes are less common. They comprise about 30% of all tornadoes, cause $< 30\%$ of tornado deaths and typically have a lifetime of at least 20 minutes, with wind speeds between 110 and 205 miles per hour. Finally, violent tornadoes are exceedingly rare, cause by far the greatest amount of fatalities, and have lifetimes that can exceed 1 hour. Tornadoes are also often rated using the Enhanced Fujita Scale, which assigns ratings from EF0 to EF5 to tornadoes. The National Weather Service Weather Forecast Office provides a breakdown of wind speed estimates by EF rating, as shown in Figure 6-2. Based on these wind speeds, we can correlate ratings EF0 and EF1 to a weak severity, EF2 through EF4 to a strong severity and EF5 to a violent severity. In the news, type EF4 tornadoes are sometimes also referred to as violent. Furthermore, since all tornadoes are dangerous and involve strong winds, unofficial news sources like status updates and blogs will often refer to weak tornadoes as strong. We attempted to build in this kind of noisiness into our own tornado dataset.

Overall, we used the above tornado characteristics in developing our rule set, so that the events occurring in our simulation were roughly similar to the attributes and events in the real world. However, we were not concerned with creating a model of high fidelity, but one that used probabilistic rules that our algorithms could reconstruct. Furthermore, we wanted our algorithms to use discovered patterns in the graph output to learn new concepts and make generalizations and predictions.

Tornado Instantiation

The tornado instantiation process started by generating a random number R between 0 and 100. Based on the number, the tornado Enhanced Fujita (EF) rating and severity are produced according to the following probability distribution:

- If R is between 0 and 69, the tornado severity is weak. If R is between 0 and 39, its rating will be EF0. If R is between 40 and 69, the tornado is assigned a rating of EF1. Based on the value R scaled by the "weak" range size ($R/69$) we choose a corresponding wind speed in the range 55-110 mph and a corresponding lifetime in the range 0-17 minutes.

- If R is between 70 and 95, the tornado severity is strong. If R is less than 85, it is designated EF2. If it is over 85, it is designated EF3. Scaling R by the "strong" range size ($R/25$), we choose the corresponding wind speed in the range 110 to 170 mph and the tornado lifetime in the range 18 to 45 minutes.

- If R is between 96 and 98, the tornado rating is EF4. With 50% probability it is designated as strong and with 50% as violent. An R value of 99 or 100 corresponds to an $EF5$ violent tornado. Based on the value R scaled by the final range size ($R/4$), we choose wind speed in the range 170 to 350 mph.

Events

Simulated tornado events can be broken up into three general categories:

1. **Tornado Description Events**

   When a tornado is created, touches down or dissipates, it generates a corresponding event. When it is created, it outputs a description that includes its rating, severity, wind speeds and all the parameters described in the previous section. When the tornado touches down it generates a description of where and when it touched down. When a tornado dissipates, it outputs a "dissipate" event and gives a summary of the damage and destruction it caused.

2. **Collision Events**

   Whenever a tornado comes in contact with a building or person, an event is generated which gives the details of the collision.

3. **Observation Events**

   When a tornado passes near a person, that person will - with some probability - make an observation about the tornado.

At the very beginning of the simulation, the model outputs a "forming" event, to indicate the time and place that the tornado would first be noticed. Shortly after, the tornado touches down, generating the corresponding event. A little after that a "tornado warning" event occurs which includes the severity of the tornado.

When a tornado comes in contact with an object, it generates either a "hit" or "crash" event depending on the severity of the tornado. If the tornado wind speed is above 100 mph, it generates the "crash" event. Otherwise, it generates the "hit" event. If the object is a type of building or structure, either a "damage" or "destroy" event is generated using the same criteria: above 100 mph a "destroy" event is generated and below 100 mph a "damage" event is generated. Because of this rule design, the simulation had a seemingly unintuitive result that if a tornado was severe enough, it did not produce any damage (only destruction). When the tornado came in contact with a person, it generated either an injury or fatality event, again depending on whether the wind speed is above 100 or not. All of the above events include numeric attributes that describe the time and place of the collision.

When a tornado is near a person, that person may make an observation. If the wind speed of the tornado is above 100, the person is likely to make an observation along the lines of "The wind is crazy out there!". Essentially, this translates to the observation containing an attribute "wind", which itself has an attribute "crazy". If the wind speed is less than 100, the person will say that the wind is "strong". If the wind speed of the tornado is above 150, the person will include the observation that there has been a power outage in the area. With a 30% probability, the bystander will observe that he or she sees debris flying through the air. These kinds of observations are qualitative characterizations of a physical event in the same way that Twitter feeds and blog posts describe physical natural disaster events using relative, qualitative terms. If we know the actual numeric value of the attribute being characterized, we can begin to associate ranges of values in a given context with categorical descriptions. Every observation also includes the time and place that it occurred at, which corresponds to the time and place tags that online posts usually have.

If the tornado has a wind speed over 160 mph, the simulation considers it strong enough to pick up an object in the model, such as a house or street sign. Whenever a tornado is in contact with a physical object, it has a 20% chance of picking up a this object at any given point in time. If the object is considered especially heavy, as in the case of office buildings, there is only a 5% chance that the tornado will be able to pick it up. If a tornado does in fact pickup an item, a "carry" event is generated indicating the time, place and type of object being carried. Once an object is in the air, there is roughly an 80% chance that it will fall out, or be dropped, at any given point in time. Thus, it is rare for an object to be carried a long distance by the tornado. When an object is dropped, the corresponding event is generated, indicating the time and place that the object was dropped onto the ground. The place that the object is dropped includes the elevation from which

the object is dropped, which allows us to examine the distribution of heights from which objects are usually dropped.

6.5.2 Summary of the Tornado Dataset

In all, the tornado dataset consisted of 7842 nodes, 15193 links and 71 unique concepts. Out of these, 22 concepts were instantiated, which allowed the algorithm to use these instances in creating clusters. For each of these 22 concepts, the algorithm averaged all known instances for that cluster to come up with a cluster "prototype". This prototype was then used as the cluster centroid or average required by the C-Means Fuzzy Clustering algorithm.

Chapter 7

Results

7.1 Little World

7.1.1 Overview

The Little World dataset was used to test the node matching and relational numeric algorithms. For the node matching algorithm, we tested whether it could accurately perform the task of entity recognition. In order to do this we added unnamed entities to the dataset and compared the unnamed or *orphan* instances to all named instances in the dataset. The algorithm then calculated the match weight between each named instance and each orphan instance. The highest matching named instance gives the most likely name for the orphan instance. For the relational numeric algorithm, we tested whether it could accurately identify which nodes had numeric attributes with consistent "greater than" patterns.

7.1.2 Node Matching

In order to test the node matching algorithm described in Chapter 5, the Little World dataset contained four unknown entities, which we attempted to accurately identify.

1. **An unnamed node, with identical attributes as named "above" nodes.**

   An "above" node contains two attributes: the two objects being described. Each of these two objects has an associated location, described using x, y and z coordinates. Essentially, this node represents the concept of "The cube is above the cylinder, where the cube is at location (x_1, y_1, z_1) and the cylinder is at location (x_2, y_2, z_2)."

   An example of a named "above" node is shown below:

```
instance above.preposition>
   attribute instance cube.3Dshape>
      attribute instance x_coordinate.coordinate 0
      attribute instance y_coordinate.coordinate 2
      attribute instance z_coordinate.coordinate 4.95
      .
   attribute instance cylinder.3Dshape>
      attribute instance x_coordinate.coordinate -2
      attribute instance y_coordinate.coordinate -2
      attribute instance z_coordinate.coordinate -3
```

2. **An unnamed node similar to an "above" node, but that only gives y and z coordinates for each of the two objects.**

3. **An unnamed node, with identical attributes to a "hit" node.**

   A "hit" node contains the following attributes: the two objects involved in the collision, the location of the collision described using x, y and z coordinates, and a velocity under 3.

   An example of a named "hit" node is shown below:

```
instance hit.contact>
   attribute instance cube.3Dshape
   attribute instance floor.physical_object
   attribute instance x_coordinate.coordinate -7.06
   attribute instance y_coordinate.coordinate 3.96
   attribute instance z_coordinate.coordinate 8.50
   attribute instance velocity.concept 1.53
   .
```

4. **An unnamed node, with identical attributes to a "crash" node. A "crash" node contains the same attributes as a "hit" node, except that the velocity is over 3.**

The node matching algorithm was run with a decay factor of 0.5, which means that the parent similarity comprises 50% of the total match weight, and the attribute similarity comprises the other 50%.

When asked to find the closest matching nodes for each of these unknown entities, the algorithm produced the results shown in Figure 7-1 using the Little World dataset.

As shown in the table, the algorithm successfully identified that the first unnamed node is actually an "above" node, since it has identical attributes to an existing "above" node. Therefore, it has a match weight of 1, or an 100% match.

For the second unnamed node, the algorithm was unable to find a perfect match, since there is no node with identical attributes in the Little World dataset. Even though the first level of attributes is the same - two 3Dshape objects - at the second level, each object only has two of the three location

Unnamed Node	Best Match	Match Weight
1	above node	1
2	above node	0.833
3	hit node, crash node	1
4	hit node, crash node	1

Figure 7-1: Node Matching Weights for Unknown Entities in Little World

dimensions: y and z. Therefore, since the node matching algorithm is recursive, it performs the following set of evaluations:

- A cube is a type of 3D shape, so its parent similarity weight is 1. This gets scaled by a factor of 0.5, since the parent similarity comprises 50% of the total match weight.
- A cylinder is a type of 3D shape, so its parent similarity weight is 1. This gets scaled by a factor of 0.5, since the parent similarity comprises 50% of the total match weight.
- A cube has two attributes: the y and z coordinates. Therefore, there is a $\frac{2}{3} \approx 0.667$ attribute similarity, when we compare to a labeled "above" node.
- A cylinder has two attributes: the y and z coordinates. Therefore, there is a $\frac{2}{3} \approx 0.667$ attribute similarity, when we compare to a labeled "above" node.

Therefore, the total match weight for the unnamed node is $0.5 \cdot (1 \cdot 0.5 + \frac{2}{3} \cdot 0.5) + 0.5 \cdot (1 \cdot 0.5 + \frac{2}{3} \cdot 0.5) \approx 0.833$. Thus, the algorithm is 83.3% confident that the unnamed node is actually an "above" node.

For each of the last two unnamed nodes, the algorithm determined that they are equally likely to be "hit" or "crash" collisions. This result makes sense since both of these types of collisions have identical qualitative attributes. The only difference between them is the magnitude of the velocity, which is a quantitative attribute. Therefore, the node matching algorithm indicates that the unnamed collisions match both "hit" and "crash" nodes with 100% confidence. This example illustrates the importance of using an algorithm that factors in numeric patterns, as well as qualitative ones. Given the two unnamed nodes and the Little World dataset, we wanted an algorithm that would notice that the velocity attributes for "hit" collisions were all numbers in a lower range than for the "crash" collisions. This would enable the algorithm to distinguish between the two qualitatively identical attributes and correctly assign each of the unnamed nodes to its respective type, using the value of the velocity. Thus, these node matching results for the Little World dataset indicated that we needed an algorithm which could detect numeric patterns and use them to perform more accurate entity recognition.

7.1.3 Relational Numeric Algorithm

The Little World dataset contained numerous instances of the "above" node which, as mentioned before, represents the concept: "Shape 1 is above Shape 2, where Shape 1 is at location (x_1, y_1, z_1) and Shape 2 is at location (x_2, y_2, z_2)." Given these attributes, we wanted the relational numeric algorithm to learn that there is a consistent "greater than" pattern associated with the concept of "above". Within the Unity Little World, the up- and-down orientation corresponds to the y-axis. Therefore, the concept of "above" can be represented as a numeric "greater than" relationship among the y-coordinates of the two objects. Specifically, we wanted the algorithm to infer that in every instance of the "above" node, the y-coordinate of the first object is greater than that of the second object. Similarly, we wanted the algorithm to learn that the concept of "above" is independent of any other orientations. Thus, we designed the Little World dataset such that there was no consistent "greater than" relationship that held for the other spatial dimensions: the x and z dimensions.

Running the numeric relational algorithm on the Little World dataset produced this desired result:

```
Greater Than Patterns Found:
1. Node Type: above.preposition
   Dimension: y_coordinate.coordinate
```

Thus, the algorithm was successful in identifying the meaning of the word "above": it represents a "greater than" numeric relationship along the y (up-and-down) orientation.

7.2 Natural Disaster Dataset

7.2.1 Overview

The results for this section are broken up into four main stages:

1. **Obtained Numeric Clusters**
2. **Cluster Averages**
3. **Clustering Algorithms Performance**
4. **Attribute Correlations**

Stage 1 describes the results of running the DBScan algorithm on each numeric dimension to discover densely packed regions or clusters. The results of this stage were then leveraged for Stages 3 and 4, which take each cluster and turn it into a numeric attribute. These numeric attributes provide additional parameters that can be used to better categorize and distinguish nodes.

Stage 2 shows some cluster prototypes, which give definitions of a cluster using attributes. The cluster prototype for a cluster C represents the "typical" node for this cluster, or how likely a node belonging to cluster C is to have each possible attribute.

Stage 3 shows the results obtained from running the Fuzzy C-Means Clustering Algorithm on both the categorical and mixed datasets. This algorithm separates the nodes into types based on their attributes. Essentially, this algorithm identifies what the type of each node should be. Therefore, this algorithm gives a measure for how well the semantic knowledge model identifies or labels an unknown node.

Stage 4 uses the results of the attribute correlation algorithm to categorize existing concepts and create new ones.

7.2.2 Obtained Numeric Clusters

This stage obtains the numeric clusters using the DBScan algorithm described in Chapter 5. The algorithm was run on the tornado disaster dataset with several different values of ϵ to determine which values produced the most conceptually meaningful clusters. ϵ, as previously mentioned, is a measure of "closeness" between two numeric values. The numeric clusters produced by this stage are then used in later stages to better group or identify nodes. Essentially, each numeric cluster is treated as a newly discovered, unnamed, concept which can filter and categorize data later on.

The natural disaster dataset contains eight different numeric types. Since several of these are measured using units that are specific to the Unity environment, we substituted real-world units for ease of understanding:

1. **Time**: Measured in minutes elapsed since the start of the simulation.

2. **Space**: Measured along the horizontal - x and z - dimensions, the Unity equivalents of longitude and latitude. For ease of understanding, we have replaced the Unity distance units with miles.

3. **Elevation**: Measured along vertical - y - dimension, the Unity equivalent of altitude, or elevation. For ease of understanding, we have replaced the Unity distance units with meters.

4. **Enhanced Fujita Tornado Scale**

5. **Tornado Path Length**: Measured in miles.

6. **Number of Buildings Damaged**

7. **Number of Buildings Destroyed**

Time Range	Size of Cluster
0 – 45 minutes	1180
50 - 51.3 minutes	5
58.5 - 62.8 minutes	18
70.7 - 72.3 minutes	4

Figure 7-2: Time Clusters for ϵ= 0.02

Obtained Numeric Clusters By Dimension

The following numeric clusters were obtained using an $\epsilon = 0.02$ and a minimum cluster size of 3. The ϵ value specifies that in order for two points to be considered close to each other, they must be no further than 2% of the total range away from each other. The minimum cluster size is specified by saying that each point P had to be close to at least 2 other points in order for these points to form a 3-point cluster. The obtained numeric clusters are shown below.

Time In the time dimension we see four clusters as shown in Figure 7-2, indicating that most events occurred during one of four time periods:

1. **0 - 45 minutes**

   This time frame corresponds to roughly the first 45 minutes from the point that the tornado is created. This cluster indicates that a large number of events happened close together for the first 45 minutes of the simulation. This makes the given time frame a relatively big one, considering that 56% of tornadoes last only 5-17 minutes, and another 28% last between 18 and 45 minutes. Therefore, at least 84% of all the tornadoes in the dataset would have been causing events that fall exclusively in this time frame, while only 16% would even last long enough to generate events that fall outside this timespan. Therefore, while this numeric cluster learns the valuable rule that most tornadoes exist - and therefore cause numerous types of events such as damage, injuries, etc - for less than 45 minutes, this result indicates that the chosen value of $\epsilon = 0.02$ is likely too large. Ideally, we would like to be able to observe time cluster patterns happening at finer levels of granularity. With this in mind, we ran our algorithms with several different values of ϵ to see if we could produce more conceptually meaningful clusters.

2. **50 - 51.3 minutes**

   This time span corresponds to 50-51.3 minutes from the start of the tornado simulation. All nodes that have a time attribute in this range must have been generated by a violent (EF4-EF5) tornado.

3. **58.5 - 62.8 minutes** This time frame represents times between 58.5 and 62.8 minutes after the start of the tornado. All nodes that have a time attribute in this range must have been generated by a violent (EF4-EF5) tornado.

	Spatial Range	Cluster Size
X Dimension	51.2 – 68.4 miles from origin	1192
Z Dimension	102.5 – 116.9 miles from origin	1192

Figure 7-3: Clusters for Horizontal Spatial Dimensions for $\epsilon = 0.02$

4. **70.7 - 72.3 minutes** This final time value cluster corresponds to times between 70.7 and 72.3 minutes after the start of the tornado, which could only be created by EF5 tornadoes, which last between 68 and 90 minutes.

Space The x and z (horizontal) space dimensions specify a location on the Unity map, in the same way that longitude and latitude would for a real world map. The y (vertical) space dimension specifies an elevation above ground level. Spatial dimension values represent the distance from the origin point (x = 0, y = 0, z = 0) in the Unity model. For ease of understanding, we replaced Unity distance units with miles for horizontal distances and meters for vertical distances.

For the horizontal dimensions, the algorithm discovered one cluster apiece for each, as shown in Figure 7-3.

- **51.2 - 68.4 miles from origin for the x-dimension**

  If we compare this range to the x dimensions of the town, which are 316 to 831 miles, we see that all of the nodes in this cluster occur within a much narrower range. This cluster represents a 17.2 mile range of x dimension values, which comprises just under 34% of the total width of the town. This indicates that most of the tornado activity was confined to a narrow space within the town, and thus caused events to occur mostly within that spatial range. Therefore, this x dimension cluster shows that, due to the nature of the tornado movement, most tornadoes caused activity in a fairly localized area.

- **102.4 - 116.8 miles from origin for the z-dimension**

  Comparing this range to the z dimensions of the town in the simulation, which are 92.8 to 164.6 miles, we see that the nodes in this cluster occur within a 14.4 mile range that is just 20% of the town length. Again this illustrates that the space covered by the most densely occurring tornado activity was confined to a fairly small length.

Combining these two numeric cluster patterns, we are able to get a clear picture of the area receiving the most tornado activity, which is equal to roughly 6.7% of the total area of the town. This could mean either that most tornadoes dissipated before they had a chance to reach further away areas or that when tornadoes reached these further away areas they had lost too much strength to cause much further activity. In either case, this pattern indicates that the tornadoes caused the most activity within a small area relative to their original position.

Elevation Above Ground Level (Y-Dimension)	Size of Cluster
0 – 16 meters	293
47 – 51 meters	45
84 – 163 meters	110
297 – 331 meters	70

Figure 7-4: Clusters for Elevation Above Ground Level for $\epsilon = 0.02$

Enhanced Fujita Scale	Size of Cluster
0	16
1	8
2	8
3	5
4	3
5	3

Figure 7-5: Clusters for Enhanced Fujita Scale for $\epsilon = 0.02$

Along the y-dimension, which represents elevation above ground level, we see four numeric clusters, as shown in Figure 7-4.

- **0 - 16 meters above ground level**

  This set of elevation values represent positions on or near the ground - since the ground is located at elevation 0 meters. As would be expected, this is by far the biggest of the numeric clusters along this dimension, which makes sense given that the majority of reported tornado activity happens on the ground.

- **47 - 51 meters above ground level**

- **84 - 163 meters above ground level**

- **297 - 331 meters above ground level**

  The other three clusters represents events that occurred at a higher elevation than any grounded object. This shows that numerous groups of events - more specifically collisions and occurrences associated with flying debris, as we will see in later sections - occurred at these elevations. Furthermore, the fact that there are three distinct clusters above ground level implies that there were three common ways in which objects ended up high off the ground - each way is associated with a unique elevation.

Enhanced Fujita Scale The interpretation of the Enhanced Fujita Scale clusters, shown in Figure 7-5, is very straightforward - each cluster represents all tornadoes of the same EF severity. The fact that the algorithm identified all five as distinct clusters is a good sign, indicating that the chosen ϵ value is small enough to group each distinct severity as its own numeric cluster.

Path Length	Size of Cluster
0.30 – 0.51 miles	3
1.04 – 1.34 miles	5
1.68 – 2.77 miles	13
3.19 – 3.60 miles	5
5.56 – 5.75 miles	5
11.68 – 11.9 miles	3

Figure 7-6: Clusters for Tornado Path Length for $\epsilon = 0.02$

Number of Buildings Damaged	Size of Cluster
0	22
2	3
5	4
7	4

Figure 7-7: Clusters for Number of Buildings Damaged for $\epsilon = 0.02$

Tornado Path Length Looking at the tornado path length clusters shown in Figure 7-6, we see the clusters distribute themselves in a roughly skewed right distribution, with the most likely path length being between approximately 1.68 - 2.76 miles. The clusters in the right tail of this distribution represent the paths of more severe or violent tornadoes, which last longer and therefore have a chance to travel further during their lifetime. We see that these longer paths usually fall into one of three "long path" categories: 3.19 - 3.60 miles , 5.56 - 5.75 miles and 11.68 - 11.90 miles. As in the case of the high-elevation clusters, these distinct ranges of values imply that there are three types of situations in which paths end up being significantly longer than the average, with each type correlated to one of these ranges. Based on our knowledge of tornado severities, we could guess that each type of situation likely corresponds to the severity of the tornado, with EF5 tornadoes creating longer paths than EF4 and EF3 ones. In further sections we show this to be true directly.

Building Damage Figures 7-7 and 7-8 show the obtained numeric clusters for building damage and destruction. The results suggest that the damaged buildings were positioned in close-together groups, such that if one building in a group was damaged, the other would be as well. This leads to certain numbers of damaged buildings being more common. Thus, we can see that there are 7 buildings closest to the starting point of the tornado, positioned in groups of 2, 3 and 4. This leads to building damage amounts of 2, 5 and 7. Similarly, we see that further out past the first 7

Number of Buildings Destroyed	Size of Cluster
0	20
7	4
10	3
11	3

Figure 7-8: Clusters for Number of Buildings Destroyed for $\epsilon = 0.02$

buildings, there is likely a group of 3 close-together buildings, making 10 the next likeliest amount of building destruction.

Looking at these building damage and destruction amounts, we see that a large portion of tornadoes seem to do minimal to no damage to buildings, as seen by the sizes of the zero clusters for both categories. Interestingly, we see a much bigger jump in value from the first to second cluster when we compare building destruction to building damage. Whereas the next common amounts of damaged buildings after 0 are 2 and 4, for building destruction we see a jump from 0 to 7 and then 10. This means that when a tornado is severe enough to actually destroy a building, it is likely to destroy many buildings and unlikely to destroy just a few. On the other hand, if the tornado is weak enough that is is only damaging, rather than destroying buildings, there is less likelihood that it will cause widespread damage.

Increasing the Level of Detail Using Smaller ϵ

In order to establish the sensitivity of the numeric clusters to the chosen ϵ value, we reran the algorithm with ϵ values of 0.015 and 0.01. This allowed us to observe the found clusters at higher levels of granularity and compare the patterns found with increased levels of detail.

With an epsilon value of 0.015, we obtained the same clusters for numeric values of Enhanced Fujita Scale and building destruction and damage, and more detailed ones for the time and spatial dimensions.

Time The time clusters obtained with an $\epsilon = 0.015$ are shown in Figure 7-9. We see that the initial range of 0 - 45 minutes that we obtained with an $\epsilon = 0.02$ actually separates out into four smaller clusters of 0 - 28 minutes, 29 - 35 minutes, 37 - 38 minutes and 40 to 43 minutes. From the size of the first cluster, it is clear that most tornado activity happens not simply in the first 45 minutes, but more specifically in the first 28 minutes. Although this is still a fairly broad range considering that over 60% of tornadoes have lifetimes of less than 28 minutes, it is an improvement over the range obtained with $\epsilon = 0.02$.

Furthermore, we can now see that past the 28 minute mark, tornado activity tends to occur most in intermittent bursts. This suggests that there are areas in which tornadoes are more likely to cause damage and other types of events to occur. Since all the tornadoes in the simulation started off in the same place and heading in the same direction, this means that each of these time periods of heightened activity likely corresponds to an area of the town with particularly densely-packed buildings or people. Therefore, by combining these time clusters with the known speed and trajectory of the tornadoes, we could in theory recreate a density map of the town.

Space For the horizontal spatial dimensions, shown in Figure 7-10, we also see a breakdown of the clusters obtained with $\epsilon = 0.02$ into smaller ones. This breakdown is not as dramatic as the

Time Range	Size of Cluster
0 – 28 minutes	1095
29 - 35 minutes	43
37 - 38 minutes	3
40 – 43 minutes	12
45 - 46 minutes	5
47 – 48 minutes	22
50 - 51 minutes	5
60 - 63 minutes	16

Figure 7-9: Time Clusters for ϵ= 0.015

Spatial Range (X-Dimension)	Cluster Size
51.2 – 65.3 miles from origin	1178
67.5 - 68.4 miles from origin	12

Spatial Range (Z-Dimension)	Cluster Size
102.4 – 104 miles from origin	16
106.1 - 116.9 miles from origin	1176

Figure 7-10: Horizontal Spatial Clusters for ϵ= 0.015

one observed for the time dimension, since each breaks down into two clusters with one still being significantly larger than the other. However, this separation does show that only a few events occur at further sections of the high-activity area - from 67.5 to 68.4 miles from the origin in the x dimension and 1061 to 1169 miles from the origin in the z dimension - while the most tornado activity is occurring at an even smaller area than the one obtained with the larger ϵ value. Furthermore, these numeric clusters suggest that the tornado paths varied more in the x dimension than the z dimension, since the range of the largest x dimension cluster is approximately 14.1 miles, while the range of the largest z dimension cluster is about 10.7 miles.

Figure 7-11 shows the elevation clusters obtained using an $\epsilon = 0.015$. As with the time dimension, we see that a decrease in the ϵ value breaks down the higher-level larger numeric clusters into more detailed, smaller ones. We can now more easily visualize the distribution of events that occur at various elevations, although the overall trend observed with the larger ϵ value remains similar. We see that the majority of events occur along the bottom of the tornado, with a smaller but still significant portion occurring at 4 times the bottom elevation range. It is interesting to note that there is a large gap between the cluster of values 14.6 - 16.3 meters and the next cluster of 29.7 - 30.9 meters. This suggests that there is likely a marked difference in the type of event that occurs at lower heights versus the type of event that occurs very far off the ground. In later sections we analyze what factors determine the elevation at which events occur.

Tornado Path Length With the path length, we see that most of the clusters remain approximately the same with an ϵ value of 0.015 as with one of 0.02. The only difference is that with the smaller value, we no longer see the cluster path length values at 11.68 - 11.9 miles. This means that this cluster consisted of values that were fairly far away from each other, and thus too sparse to be considered a cluster when smaller measures of "closeness" are used. This could suggest one

Elevation Above Ground Level (Y-Dimension)	Size of Cluster
0 – 16 meters	293
47 – 51 meters	45
84 – 103 meters	35
109 – 117 meters	13
124 – 140 meters	53
146 – 163 meters	9
297 – 309 meters	52
315 - 331	18

Figure 7-11: Elevation Clusters for ϵ= 0.015

Path Length	Size of Cluster
0.30 – 0.45 miles	3
1.13 – 1.34 miles	4
1.68 – 2.77 miles	13
3.30 – 3.60 miles	5
5.56 – 5.75 miles	5

Figure 7-12: Path Length Clusters for ϵ= 0.015

of several things: (1) that tornadoes which *do* last long enough to create paths of this length vary widely in behavior and speed of motion, leading to widely differing longer path lengths (2) that the lifetimes of long-lasting tornadoes vary widely (3) that the dataset contains an insufficient number of data points of long-lasting tornadoes to create dense clusters for longer path lengths. Since we tried to adhere to the real-world observed proportion of weak to violent tornadoes, the last option is likely - compared to the number of weak tornadoes in our dataset the number of violent ones is small.

Further Increased Level of Detail Using Smaller ϵ Finally, we ran the algorithm with an ϵ value of 0.01. This produced nearly identical results as with the value 0.015, with slightly more fragmentation along the time and elevation dimensions. Figure 7-13 illustrates this successive breakdown of clusters into smaller ones for the elevation dimension. These results suggest that while lower values of ϵ could perhaps provide a more detailed fine-grained picture of the distribution of numeric values, they would likely become too small, and fragment the data too much, to produce conceptually meaningful clusters.

Combining the Numeric Dimensions with Qualitative Data

It should be noted that while we learned much from examining these numeric clusters by themselves, the amount of information they can give us is limited because each numeric dimension and each cluster gives us only relative frequencies for one type of value at a time. In later sections we explore how each of these clusters is correlated with other attributes, as well as with each other, which

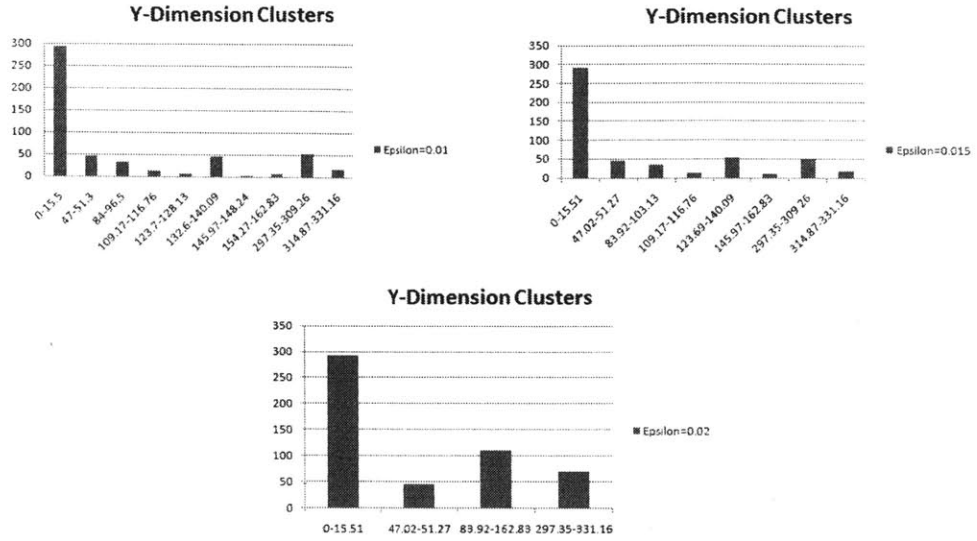

Figure 7-13: Y-Dimension Clusters for Varying Values of Epsilon

provides a much richer and more meaningful picture of the patterns and connections in our dataset.

7.2.3 Cluster Averages

Since this algorithm was run twice - once on purely categorical data and once on mixed categorical and numeric data - each run had its own set of cluster prototypes. In the mixed data case, the cluster prototypes describe the learned numerical attributes that are typical for each cluster, in addition to the qualitative ones.

Below are sample cluster prototypes with a decay factor of 0.4 and an ϵ of 0.015. As mentioned earlier, the decay factor represents the confidence with which observed semantic patterns are propagated to higher levels of abstraction. ϵ is a measure of closeness between points - the smaller the ϵ value, the closer together points have to be to belong to the same cluster.

Tornado Warning Cluster The tornado warning cluster prototype is shown in Figure 7-14. Each row in this output shows (1) a tornado warning attribute (2) the average likelihood that a tornado warning will have that particular attribute. Thus, the first row shows that the likelihood that a tornado warning will warn about a *weak* severity tornado is 54.76%.

It is important to note that the prototype includes both (1) attributes that were instantiated and (2) attributes that were *not* instantiated, but whose children were. For those attributes that were instantiated, the likelihood represents the actual proportion of tornado warning instances that had this attribute. In other words, out of all the instance of tornado warnings in the dataset, 54.76% of them had an attribute *weak.severity*. For attributes which were *not* directly instantiated, but had

Attribute	Weight
weak.severity	0.5476
strong.severity	0.3095
violent.severity	0.1429
severity.attribute	0.4
attribute.0	0.16
minute.time_unit	1
time_unit.time	0.4
time.concept	0.16
concept.entity	0.064
entity.0	0.0256

Figure 7-14: Tornado Warning Cluster Prototype - Qualitative Attributes

children that were, the likelihood is decayed by a factor of 0.4 for each level that you move further away from the instantiated child. Thus, if we add up the probabilities of all the different possible severities - weak, strong and violent - we see that $0.5476 + 0.3095 + 0.1429 = 1$, which means that *all* tornado warnings had one of these three severities. Thus the weight assigned to the *severity* parent attribute is $1 \cdot (Decay\,Factor) = 1 \cdot 0.4 = 0.4$. Thus, the decay factor represents a way to quantify the confidence with which observed patterns propagate to higher levels of abstraction. Essentially, the decay factor answers the question "So far, every tornado warning I've seen has had some severity associated with it. How confident am I that every tornado warning I will *ever* see will have a severity associated with it?" The higher the decay factor, the more freely the algorithm makes generalizations.

Similarly, this description shows that every tornado warning has a minute timestamp that identifies when the warning was issued. This can be seen from the fact that the prototypical tornado warning has an attribute *minute* with probability 1, or 100% of the time. Based on this fact, the algorithm infers that a tornado warning may have *any* timestamp or time associated with it - which is reflected in the weights of the attributes *time_unit* (0.4) and *time* (0.16). We see here again that as we move up the hierarchy, the confidence with which the algorithm "learns" decreases. Thus, the algorithm is 40% confident that a tornado warning will have a timestamp with *any* time unit e.g. *time_unit.time*, but only 16% confident that a tornado warning can have any measure of time associated with it. In other words, the algorithm is assessing the likelihood that the time of a tornado warning can be expressed in units other than minutes, or indeed, without units at all, but using any arbitrary measure of time.

Since the natural disaster dataset always uses the "minutes since the start of the simulation" time unit to timestamp, this makes sense. If we were given a different dataset in which many different types of timestamps were used for a tornado warning - for instance minutes, hours, military time, AM/PM time - the algorithm's confidence that this time measure would specifically be in minutes would decrease. In other words, the weight of the attribute *minute.time_unit* would decrease from

Numeric Attribute Name	Range	Weight
minute.time_unit	0 – 28 minutes	1

Figure 7-15: Tornado Warning Cluster Prototype - Quantitative Attributes

1. The smaller the portion of tornado warnings that use the minutes timestamp, the smaller the weight of *minute.time_unit*. Yet, as this portion would grow smaller, the weight of *time_unit.time* would stay exactly the same, since no matter what specific time unit was used, every instance would still have a timestamp with *some* time unit. From this pattern, it becomes clear that as the dataset becomes more heterogeneous, it becomes easier and easier for the algorithm to learn at higher levels of abstraction. As we begin to use increasingly diverse types of time measurements, the probability of using any specific one gets increasingly smaller relative to the probability of using *any* time unit - which holds steady at 0.4.

When we add in the numeric data, the algorithm adds a numeric attribute to the tornado warning prototype, shown in Figure 7-15. This output says that all tornado warnings have a numeric attribute of "a time between 0 and 28 minutes since the start of the simulation". If we look back at the previous section on numeric clustering, we see that this was one of the numeric time clusters we obtained from detecting pattern along the time dimension. The fact that this attribute has a weight of 1 indicates that 100% of all tornado warnings were issued in this time frame - in other words, close to the start of the tornado simulation - within the first 28 minutes. Essentially, this numeric attribute represents the algorithm learning that tornado warnings are typically issued closer to the start of the tornado, which is both a common sense and useful pattern to learn. The algorithm can then use this information to predict with higher confidence whether some unknown event is, in fact, a tornado warning: if this unknown event happens toward the end of the tornado's lifetime, it is less likely to be a tornado warning.

Drop Events The output in Figure 7-16 output shows the categorical cluster prototype for all instances of a tornado dropping something. This result shows that all such instances were labeled with a time and location (along the x, y, and z dimensions). Furthermore, we see that 82.6% of the time that the tornado dropped something, it was a house, and the other 17.4% of the time it was a shed. Since a tornado can only drop something it has previously picked up and carried, this also means that 82.6% of the time that a tornado carried something it was a house, and the other 17.4% of the time it was a shed. This relates directly to the proportion of structures used in the simulation. Since the simulation contained only three kinds of buildings: houses, office buildings and sheds, of which houses were by far the most numerous, and office buildings were the heaviest, these statistics make sense. This data indicates that while some tornadoes were strong enough to carry and drop houses, none were powerful enough to carry and drop an entire office building. Similarly,

Attribute	Weight
house.building	0.8256
shed.building	0.1744
building.structure	0.4
structure.entity	0.16
entity.0	0.064
x_coordinate.coordinate	1
y_coordinate.coordinate	1
z_coordinate.coordinate	1
coordinate.location	0.4
location.entity	0.16
entity.0	0.064

Figure 7-16: Drop Event Cluster Prototype - Qualitative Attributes

Numeric Attribute Name	Range	Weight
x_coordinate.coordinate	51.2 – 65.3 miles from origin	1
z_coordinate.coordinate	106.1 – 116.9 miles from origin	1
y_coordinate.coordinate	0 – 15.51 meters above ground	0.222
y_coordinate.coordinate	47.02 – 51.27 meters above ground	0.1333
y_coordinate.coordinate	83.92 – 103.13 meters above ground	0.1556
y_coordinate.coordinate	123.69 – 140.09 meters above ground	0.1556
y_coordinate.coordinate	145.97 – 162.83 meters above ground	0.1556
y_coordinate.coordinate	297.35 – 309.26 meters above ground	0.0444
y_coordinate.coordinate	314.87 – 331.16 meters above ground	0.0889

Figure 7-17: Drop Event Cluster Prototype - Quantitative Attributes

the attribute weights show that a tornado is much more likely to carry and drop a house than a shed - which we know to be reasonable since there were many more houses than sheds in the simulation.

Adding in the numeric data, we obtained additional numeric attributes for the drop event, shown in Figure 7-17. Looking at the horizontal spatial dimensions, we see that all drop events occurred within the previously discovered space of high tornado activity: in the x value range of 51.2 - 65.3 miles from the origin and the z value range of 106.1 - 116.9 miles from the origin. On the other hand, the distribution of elevations from which objects were dropped by a tornado varies greatly. Graphing this distribution of elevations, we obtain a detailed picture of the varying heights from which a tornado is likely to drop objects: Figure 7-18. Based on this fairly even distribution we gain the understanding that there is no one "typical" height from which a tornado is likely to drop an object, although the likelihood seems to decrease slightly as you move higher. Furthermore, we know that a tornado is unlikely to carry objects any higher than an elevation of approximately 331 meters.

Tornado Cluster Since tornadoes contained by far the most attributes of any type of node in our dataset, the prototype tornado was by far the longest. The most informative attribute weights

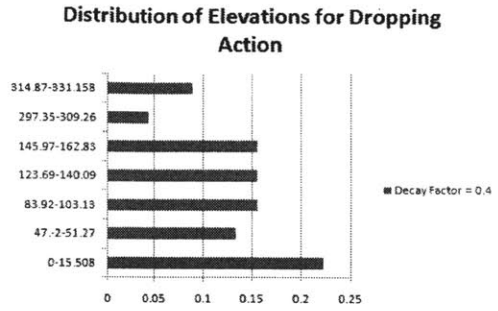

Figure 7-18: Distribution of Dropping Action by Elevation Along Y-Dimension

Attribute	Weight
carry.event	0.3226
drop.event	0.3226
damage.event	0.7484
destruction.damage	0.6452
kill.damage	0.6452
dissipate.event	1
wind.weather	1
weak.severity	0.6452
strong.severity	0.4194
violent.severity	0.1613
severity.attribute	0.4
enhanced_fujita_scale.tornado_scale	1
building_damage.building_statistic	1
building_destruction.building_statistic	1
path_length.disaster_statistic	1
minute.time_unit	1
x_coordinate.coordinate	1
y_coordinate.coordinate	1
z_coordinate.coordinate	1

Figure 7-19: Tornado Cluster Prototype - Qualitative Attributes

are shown in Figure 7-19.

This cluster prototype does a very good job of recreating the rules and probabilities that were built into the simulation. It demonstrates that all tornadoes were tagged with time, location, path length, number of buildings destroyed and damaged, wind and an Enhanced Fujita severity rating.

Furthermore, it states that any given tornado is 34.4% likely to carry and drop an object; or stated alternatively, that 34.4% of tornadoes carry and drop things. Similarly, a tornado is 74.8% likely to cause damage and 64.5% likely to cause destruction.

Finally, combining explicit tagging of tornado category with subjective human observations of tornadoes, we obtain the knowledge that tornadoes are 64.5% likely to be called violent, 41.9% likely to be called strong and 16.1% likely to be called weak. It is important to note that these probabilities clearly add up to more than 100%. This is caused by the fact that the same tornado may be called

Numeric Attribute Name	Range	Weight
enhanced_fujita_scale	0 – 0	0.29
enhanced_fujita_scale	1 – 1	0.258
enhanced_fujita_scale	2 – 2	0.1613
enhanced_fujita_scale	3 – 3	0.1613
enhanced_fujita_scale	4 – 4	0.0645
path_length	0.30 – 0.51 miles	0.0968
path_length	1.04 – 1.34 miles	0.0645
path_length	1.68 – 2.77 miles	0.29
path_length	3.30 – 3.60 miles	0.0967
path_length	5.56 – 5.75 miles	0.1613

Figure 7-20: Tornado Cluster Prototype - Quantitative Attributes

strong by one source and violent by another by the rules of our simulation. The severity descriptions of tornadoes are generated in two ways. First, the tornado is assigned a severity at the start of the simulation - this corresponds to the rating that a meteorologist would give. Second, subjective observations are made by bystanders - this corresponds to a social networking post such as "Wow! That's a strong tornado!". Thus, since an observer may ascribe a subjective "strong" severity to a weak or violent tornado, as opposed to tornado experts, there is some intentional overlap and disagreement in the descriptions of the tornadoes.

When we add in the numeric attribute weights observed for tornadoes, we obtain an even more detailed picture. Several informative numeric attribute weights are shown in Figure 7-20.

Looking at the first five numeric attributes, we see that each of the quantitative Enhanced Fujita (EF) severity ratings has been converted into a tornado attribute. This means that each EF rating cluster should be thought of as a unique concept that can be used to distinguish one tornado from another. This is exactly the kind of inference that we wanted our algorithm to make: if a certain range of numeric values occurs often enough, it should have a corresponding qualitative label. And indeed, in the same way that the frequently occurring value 3.14159265 was labeled as the concept π, the frequently occurring EF value of 0 is labeled as the concept "EF0".

The numeric path length attributes give a rough distribution of the path lengths for tornadoes. This distribution indicates that the most common path length for a tornado is between 1.68 and 2.77 miles, with 29% of tornadoes having path lengths in this range. As we will see in later sections, these shorter path lengths are typically associated with weaker tornadoes. Similarly, the second most common path length was between 5.56 and 5.75 miles, with approximately 16% of tornadoes having path lengths in this range. These longer path lengths correspond to stronger tornadoes, which have longer lifetimes and consequently have the time to travel further from their starting point. In later sections, we will see this correlation between path length and tornado lifetime directly.

Numeric Attribute Name	Range	Hit Event Weight	Crash Event Weight
minute	0 – 28 minutes	1	0.943
minute	29 – 35 minutes	0	0.0316
minute	47 – 48 minutes	0	0.0127
minute	60 – 63 minutes	0	0.0127
x_coordinate	51.2 – 65.3 miles from origin	1	0.9873
x_coordinate	67.5 – 68.4 miles from origin	0	0.0127
y_coordinate	0 – 15.51 meters above ground	0.5625	0.4367
y_coordinate	47.02 – 51.27 meters above ground	0.1071	0.1392
y_coordinate	83.92 – 103.13 meters above ground	0.0446	0.1139
y_coordinate	109.17 – 116.76 meters above ground	0.0446	0.0506

Figure 7-21: Hit and Crash Events Cluster Prototypes - Quantitative Attributes

Hit and Crash Event Clusters For the "hit" and "crash" collision events, the cluster prototypes produced showed very similar qualitative attribute weights, but several significantly different numeric attribute weights. This illustrates the idea pointed out by the results of the node matching algorithm used on the Little World dataset: the concepts "hit" and "crash" are qualitatively identical - the difference between them is a purely numeric one. In other words, a "crash" is simply a stronger version of a "hit". Because of this, the purely qualitative node matching algorithm used on the Little World dataset was unable to distinguish between the two concepts. The cluster averages, on the other hand, distinguish between these two concepts more easily based on their numeric attributes. Several of these numeric attributes are shown in Figure 7-21.

Comparing the numeric time attributes, we see that every single "hit" collision occurred in the first 28 minutes, while 5.7% of "crash" collisions occurred after that. This is due to the fact that only severe tornadoes are strong enough to cause "crash" collisions, and only severe tornadoes have lifetimes long enough to last more than 28 minutes. Thus, any collision that occurred after the 28 minute mark, can be immediately identified as a "crash" collision, and *not* a "hit" collision.

Similarly, only "crash" collisions occur at further away distances along the x dimension e.g. between 67.5 and 68.4 miles away from the origin. This is due to the fact that only severe tornadoes last long enough to travel that far away from the starting point. Thus, any collision that occurs at this distance can immediately be identified as a "crash" and not a "hit".

Along the elevation dimension we see a similar pattern emerge. Since severe tornadoes are more likely to pick up and carry objects, more of the "crash" collisions - which can only be produced by severe tornadoes - are seen at higher elevations. Whereas approximately 56% of all "hit" collisions occur close to the ground, only about 44% of "crash" collisions occur there. As the elevation increases, it becomes more likely that the collision was produced by a more severe tornado. 14% of crashes occur between 47 and 51 meters off the ground, while 11% of hits occur at this elevation. 11.4% of crash collisions occur between elevations of 83.9 and 103 meters off the ground, while less

than 4.5% of hit collisions occur at this height.

Combining all the numeric attribute differences between these two types of collisions, it becomes significantly easier to distinguish between "hit" and "crash" events.

Conclusion The obtained cluster prototypes give precise and detailed definitions of each type of concept in the tornado dataset. From examining the weights of each attribute, we gain an understanding of (1) which attributes are typically associated with a particular concept (2) exactly how likely the concept is to have each attribute. Furthermore, by observing which higher-level attributes have high weights, we detect higher-level semantic patterns for each concept. For instance, we not only know which particular severities are associated with a tornado, but also that *all* tornadoes have severities. And we make this kind of inference using a bottom-up approach - we are not *given* a rule "All tornadoes have severities" for the tornado concept, we learn it by examining all known instances of tornadoes.

The use of numeric attributes, created from dense value clusters detected by the DBScan algorithm, gives an even more precise characterization of a concept. Numeric attributes allow us to associate ranges of values with qualitative concepts. In the same way that we would consider heights over 6 ft 5 in unusually tall for a man, the tornado cluster prototype considers path lengths between 0.3 and 0.51 miles unusually short for a tornado. This demonstrates a much more detailed understanding of typical tornado characteristics than a purely qualitative definition could give.

7.2.4 Clustering Algorithms Performance

In order to test the performance of the Fuzzy C-Means clustering algorithm, we ran it on the tornado dataset with values of ϵ ranging from 0.01 to 0.02 and decay factors ranging from 0.3 to 0.6. We tested the correctness of the results by comparing the placement of each point to its node type. Thus, if an instance of type "tornado" is placed in the "tornado" cluster by the C-Means algorithm, this is considered a correct placement. Because each point in the dataset belongs to exactly one cluster, placement into any other cluster is considered incorrect. This is an overly strict measure of correctness since several of the different clusters contain identical attribute types, and could therefore be considered parts of the same cluster. However, this allowed us to see if adding in the numeric data allowed the algorithm to distinguish between qualitatively identical nodes based on discovered numeric patterns.

As described in Chapter 5, the Fuzzy C-Means clustering algorithm outputs a U matrix where the rows represent clusters C_1 to C_n and the columns represent points P_1 to P_m. Each cell value $U(C_j, P_k)$ represents the degree to which the point P_k matches, or belongs, to cluster C_j. Therefore, there are two ways in which this matrix could be used to group points into clusters.

The first way we could choose some cutoff match weight U_{match} and say that any point P_k whose

$U(C_j, P_k) > U_{match}$ should be placed in cluster C_k. In order to make this method more flexible, one could choose a U_{match} on a cluster-by-cluster basis. For instance, one could decide that the top X number of match weights for cluster C_k should be accepted, and anything below those rejected. Suppose we chose to take the top *two* weights for a cluster where (1) 3 points matched with weight 0.8 (2) 2 with 0.7 and (3) five with 0.3. We would set the cutoff $U_{match} = 0.7$ take the top five points and reject the bottom five.

Alternatively, instead of going through the U matrix cluster by cluster, we could choose to go through point-by-point, and for each one say that it *must* belong to the top matching clusters. This ensures that a point has to belong to at least one cluster. It also guarantees that when a point is equally likely to belong to several different clusters, it is placed in all of them. We could make this method even more flexible by establishing some metric that allows us to add points to all clusters for which its U_{match} is "close enough". For example, if a dataset has 5 clusters, and a given point P_k has match weights $U(P_k, C_1) = 0.9$, $U(P_k, C_2) = 0.9$, $U(P_k, C_3) = 0.8$, $U(P_k, C_4) = 0.2$, $U(P_k, C_5) = 0.2$, it would be reasonable to suppose that the point should belong to C_3, even though the match weight is lower than for C_1 or C_2. Thus, we would deem a match weight of 0.8 to be "close enough" based on the fact that the match weight drops off dramatically below that match value, all the way to 0.2. This would require some sort of measure for what constitutes a sharp drop off. For datasets in which a point is likely to belong to many different clusters, this approach seems like a reasonable one. However, for the purposes of our dataset, in which every point belongs to only one cluster, we used the simplest version of this method - placing a point P_k only in the top matching clusters.

In practice, we observed that the cluster threshold method was not an effective way of grouping points into clusters. This was due to the fact that match values varied significantly across clusters. For some portion of the clusters, the set of attributes that defined the cluster prototype were highly unique, and thus the match weights were very high for points that belonged and very low for points that did not. However, for other less unique clusters, match weights were much closer together - a point that belonged to a cluster could have a match weight of 0.6 and a point that did not belong could have a match weight of 0.55. Furthermore, for clusters in which the data points were more highly heterogeneous it was even harder to determine whether a given point belonged to a cluster, since the standard deviation from the "prototypical" cluster average was higher. This made it practically impossible to choose any consistent threshold or cutoff value, even on a cluster-by-cluster basis. Therefore, we ultimately chose the second point-by-point method as the way to use the U matrix to assign points to top matching clusters.

Figure 7-22 shows the number of points correctly identified and the percent success rate of the C-Means clustering algorithm on both the purely qualitative and mixed natural disaster dataset with an $\epsilon = 0.02$. In this table, the success rate is measured by the number of points whose top match was

Decay Factor		Categorical Data		Mixed Data	
0.3		1369 / 1725	79.36 %	1442 / 1725	83.59 %
0.4		**1375 / 1725**	**79.71 %**	**1448 / 1725**	**83.94 %**
0.45		1356 / 1725	78.61 %	1432 / 1725	83.01 %
0.5		1356 / 1725	78.61 %	1428 / 1725	82.78 %
0.6		1282 / 1725	74.32 %	1426 / 1725	82.67 %

Figure 7-22: Success Rate of Fuzzy C-Means Clustering Algorithm with $\epsilon = 0.02$

Decay Factor		Categorical Data		Mixed Data	
0.3		1369 / 1725	79.36 %	1420 / 1725	82.32 %
0.4		**1375 / 1725**	**79.71 %**	1420 / 1725	82.32 %
0.45		1356 / 1725	78.61 %	**1424 / 1725**	**82.55 %**
0.5		1356 / 1725	78.61 %	1420 / 1725	82.32 %
0.6		1282 / 1725	74.32 %	1418/ 1725	82.20 %

Figure 7-23: Success Rate of Fuzzy C-Means Clustering Algorithm with $\epsilon = 0.015$ and $\epsilon = 0.01$

the "correct" cluster. These results show that adding in the numeric attributes slightly improves the ability of the algorithm to correctly place nodes into their clusters - by 4.23%. Furthermore, it shows that the highest success rate given an $\epsilon = 0.02$ is obtained when the decay factor is equal to 0.4.

For values of $\epsilon = 0.015$ and $\epsilon = 0.01$, we also see that running the Fuzzy C-Means clustering algorithm on the mixed dataset performs better than on the categorical dataset. The success rates were identical for both ϵ values; they are shown in Figure 7-23. However, for these two values a better success rate is obtained using a decay factor of 0.45, instead of 0.4. If we compare overall performance across all ϵ values, the highest success rate (83.94%) is obtained using $\epsilon = 0.02$ and a decay factor is equal to 0.4.

To further compare the algorithm's performance on the two datasets, we applied a stricter definition of "correctness". In the previous definition, if a point had tied top matching clusters, we considered the placement correct if any one of those top matches was correct. Thus, if a point P_k had highest matching values $U(P_k, C_1)$,$U(P_k, C_2)$ and $U(P_k, C_3)$, its placement was considered correct if any one of the clusters C_1, C_2 or C_3 was the correct choice. By applying a stricter definition, we said that a point cluster assignment was only correct if it *only* had a top matching value for the correct cluster. Using this definition, the success rate of the Fuzzy C-Means Clustering algorithm was significantly lower - 44.40% at best. Figure 7-24 shows the revised success rates for both the categorical and mixed datasets. Thus, the use of numeric attributes allowed the clustering algorithm

Decay Factor		Categorical Data		Mixed Data	
0.3		750/1725	43.48%	1442/1725	83.59%
0.4		**766/1725**	**44.40%**	**1448/1725**	**83.94%**
0.45		747/1725	43.30%	1432/1725	83.01%
0.5		747/1725	43.30%	1344/1725	77.91%
0.6		601/1725	34.84%	1426/1725	82.67%

Figure 7-24: Strict Success Rate of Fuzzy C-Means Clustering Algorithm with $\epsilon = 0.02$

to be significantly more accurate in assigning points to *only* the correct cluster - by 39.54%. Based on the large difference between success rates and strict success rates, there is clearly a large portion of data points that are qualitatively identical. Yet, when we add in numeric attributes based on discovered numeric patterns, these points become different from one another. This allows them to be placed in only the correct cluster, which exhibits the same numeric attributes as the point.

Taking a closer look at the clusters that most often tied for top match values, we see the following groups:

```
crash, hit
carry, destruction
injure, kill
tornado_warning, dissipate
```

In each group the pair of node instances have identical qualitative attributes. For instance, in the case of crash events and hit events, both have attributes of:

- The building that was hit or crashed into.
- The time of the collision.
- The x, y, and z coordinates of the collision.

Since a crash event can only occur if the tornado is severe enough to cause a crash - instead of just a hit - a portion of the crash events would have numeric attributes that were unique to severe tornadoes. For instance, since only severe tornadoes last longer than 28 minutes, any crash event that had a time after the 28 minute mark, would have a numeric attribute that reflects that fact. Similarly, any crash event that occurred far away from the tornado starting point, and would thus have the corresponding x and z dimension values, would have numeric attributes that showed that. On the other hand, no hit events would have such numeric attributes, since they are only produced by weaker tornadoes. Thus, the mixed data run of the C-Means clustering algorithm would be able

to accurately distinguish between a portion of the crash events that the categorical data run would not.

For all values of c and decay factor, clustering on mixed data outperformed clustering on categorical data. Depending on which ϵ value was chosen, the numeric clusters that are used to make numeric attributes change, and thus the performance of the C-Means clustering algorithm changed.

The highest success rate was consistently obtained using an ϵ value of 0.02. As we saw in the previous section, this value of ϵ creates more generalized, or coarser granularity, numeric clusters on certain dimensions. This implies that there is somewhat of a trade off between the value that enables the algorithm to best determine the correct type - or cluster - of a node and the value that enables the semantic knowledge representation to discover new patterns and concepts. This makes sense since the C-Means clustering algorithm performs the best when the data points within each cluster are the most homogeneous, which allows it to more easily group them together and discover similar points. On the other hand, in order for an algorithm to discover new patterns and variation *within* a cluster it needs a finer level of granularity that allows it to differentiate between nodes within a cluster. Therefore, it is in the best interest of a by-cluster pattern-finding algorithm to make the points in a cluster look as heterogeneous as possible. With this in mind, the sub-clusters algorithm in the following section was run with an ϵ value of 0.015.

By varying the decay factor, we determined that the decay factor which gave the highest success rate for the tornado dataset was 0.4. This means that in assigning attribute weights to concepts, we generalized to each higher level of abstraction with 40% confidence.

Results Summary

Running the Fuzzy C-Means clustering algorithm on both qualitative and mixed data, we observed that the addition of numeric attributes significantly improved the performance of the algorithm. If we use a more lenient measure of correctness that only looks at whether *one* of the top-matching clusters for a point is correct, the mixed data algorithm slightly outperforms the qualitative data algorithm by 4.23% (83.94% vs. 79.71%). Using a stricter measure of correctness - that *every* top-matching cluster must be correct - the mixed data algorithm outperforms the qualitative data algorithm by 39.54%. This result highlights the fact that the addition of numeric attributes allows the algorithm to distinguish between qualitatively identical concepts. In other words, by observing differences in the values associates with each concept, the algorithm is able to discover distinguishing characteristics.

In order to empirically determine the optimal parameter values for the decay factor and ϵ, the Fuzzy C-Means Clustering Algorithm was run with five different decay factor values, ranging from 0.3 to 0.6, and three different c values ranging from 0.01 to 0.02. The highest overall success rate was observed with a decay factor of 0.4 and an $\epsilon = 0.02$. These values are functions of the particular

dataset being used.

Given a different dataset with more levels of detail, we would expect to see a higher optimal value for the decay factor. This is due to the fact that the confidence with which we generalize to higher levels of abstraction is directly related to the amount of information lost each time we move up a level. The more information is lost per level, the less confident we are that the observed patterns will hold at the next level up. Thus, the more levels a dataset has, the less information is lost per level, the higher the decay factor.

The optimal value of ϵ depends on the homogeneity of the instances for each concept. For the purposes of entity identification, we want the instances of a cluster to have as similar attributes as possible, so that we can more easily identify them. Choosing an ϵ value that is too small will lead to a larger number of different numeric attributes. The higher the number of numeric attributes, the more likely it is that instances in the same cluster will have different numeric attributes. This increases heterogeneity within the cluster and makes it more difficult to group instances by common attributes.

7.2.5 Sub-clustering

As described in Chapter 5, the sub-clustering algorithm detects attribute correlations for a given concept. These correlations can then be used to subdivide a concept into sub-concepts, where each sub-concept has more closely correlated attributes than the parent concept.

Thus, if we see that within the concept of "animal" there is a subgroup with closely correlated attributes "wings" and "beaks" and another subgroup with closely correlated attributes "fur" and "mouth", we can subdivide the "animal" cluster into two sub-clusters of "bird" and "mammal". This process does not require the algorithm to *know* the sub-cluster labels "bird" and "mammal" - it simply notices the attribute patterns which can then be used to create unnamed sub-clusters. In this way, the algorithm can learn concepts that did not exist in the dataset, which makes it a particularly powerful learning technique.

For the tornado dataset, the sub-clustering algorithm discovered attribute correlations for 8 of the 22 clusters. A selection of these attribute correlations, and their implications, are described in the following sections.

Tornado Warning Applying the attribute correlating algorithm to nodes of type *tornado warning*, we learned the following simple rules:

- If a tornado warning has a weak severity, it will not have a strong or violent severity.
- If a tornado warning has a strong severity, it will not have a weak or violent severity.
- If a tornado warning has a violent severity, it will not have a weak or strong severity.

Since a tornado warning can only have one severity, all three of these rules are true and illustrate the idea of mutual exclusivity: tornado warning severities "weak", "strong" and "violent" are mutually exclusive. Throughout the rest of the clusters, a fraction of the rules found were similar to this one: they pointed out which sets of attributes were mutually exclusive. This highlights the fact that the algorithm is able to discover and recreate the rules used to build the dataset, which is an excellent indicator that it is performing correctly. Furthermore, it allows the algorithm to make stronger generalizations, since it can automatically rule out a large subset of impossibilities quickly. For instance, as we will see further in this section, a tornado being violent has a whole host of other implications that affect its path length, level of destruction, etc. If we know that a tornado is weak, we then automatically know that a tornado is not violent and therefore does not have a path length or level of destruction that is associated with violent tornadoes. This is precisely the desired behavior we described in Chapter 3 on negation, which is the idea that the existence of node A inhibits the possibility of the existence of node B in some particular context. Finally, this property of mutual exclusivity allows us to determine that the cluster of tornado warnings can - and should - be divided into sub-clusters corresponding to weak tornado warnings, strong tornado warnings and violent tornado warnings. Thus, the attribute correlation algorithm enables our semantic knowledge representation to learn three new sub-categorizations of tornado warning; stated another way, the algorithm enables the representation to learn three new concepts that are children of the *tornado_warning* concept.

Carry Action For the *carry.action* node, we obtain the following rules:

- If a carry action occurred between times 47 and 48 minutes from the start of the simulation, then the action likely occurred between x dimension distances 67.56 and 68.44 miles away from the origin, and vice versa.
- If a tornado carried a shed, this action likely occurred within the first 28 minutes.
- A tornado carrying a shed and a house are mutually exclusive e.g. a tornado carries only one object per carry action.

An important point to observe for this and following rules is that each rule named here is the condensed conceptual version of the set of rules outputted by our algorithm. Thus, the first rule is actually delineated more formally by the algorithm to produce the "if and only if" rule shown above. The precise output generated by our algorithm looks more like this:

- If a carry action occurred between times 47 and 48 minutes, then the action likely occurred between x dimension locations 67.6 and 68.44 miles away from the origin.
- If a carry action occurred between times 47 and 48 minutes, then the action likely did not occur between x dimension locations 51.26 and 65.32.

- If a carry action occurred within the first 28 minutes, it likely did not take place between x dimension locations 67.56 and 68.44.
- If a carry action took place between x dimension locations 67.56 and 68.44, it likely occurred between times 47 and 48 minutes.
- If a carry action took place between x dimension locations 67.56 and 68.44, it likely did not occur within the first 28 minutes.

From this more precise set of rules we see that the algorithm goes through numerous possible variations of the rule. It is especially important to delineate when the rule differs most from the "expected" overall probabilities. Since the cluster of x dimension values between 51.26 and 65.32 miles is by far the most densely packed and largest cluster of x dimension values, it is particularly noteworthy to point out that carry actions occurring between times 47 and 48 minutes do *not* occur in this range of x values. Given that a randomly chosen event would have a very high probability of taking place in this x value range, since $\frac{1178-12}{1178} = 98.98\%$ of clustered events occur in this range, this is a particularly strong statement. Similarly, the overwhelming majority of events occur within the first 28 minutes, and therefore the rules state explicitly that a carry action that takes place between x locations 67.56 and 68.4 is highly unlikely to occur between these times.

Thus, from this rule we gain the understanding that the carry actions that occurred furthest away from the starting point of the tornado occurred at much later times than the majority of events. Since the range of times 47-48 minutes after the start of the tornado is a very specific one, we get a precise picture of the time and location of this set of carry actions.

From the second rule shown above, we learn that the shed must be located in a place that allows a tornado to consistently reach it and pick it up in 28 minutes or less. Thus, this rule allows us to learn about the original location of the shed: it was likely located near the starting point of the tornado, since it is only carried by a tornado in the first range of time values e.g. close to the touchdown time of the tornado.

Therefore, these rules allow the semantic knowledge representation to learn that the within the cluster of carry actions, there are likely three sub-clusters: (1) the sub-cluster of carrying actions that occurred between times 47 and 48 minutes from the start of the simulation and x locations 67.56 to 68.44 miles away from the origin (2) the sub-cluster of carrying actions in which a shed was carried - these will all have been carried within the first 28 minutes (3) the Sub-cluster of carrying actions in which a house was carried - these will all have been carried *after* the first 28 minutes. Thus, the semantic knowledge representation has learned three new concepts that are children of the *carry.action* concept. If we were to give each of these new sub-cluster concepts a name, we would likely call them something like (1) "far away carrying action", (2) "carrying a shed action", (3) "carrying a house action".

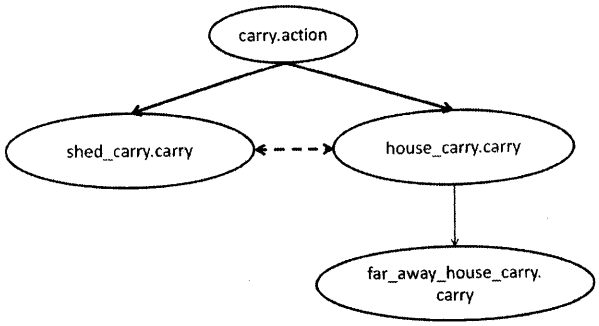

Figure 7-25: Learned Sub-cluster Hierarchy for the Carry Action

It is important to note here that the rules of mutual exclusivity we have learned in this case make it impossible for a carry action to belong to both sub-clusters "carrying a shed action" and "carrying a house action", or both sub-clusters "far away carrying action" and "carrying a shed action". In the second case, we have learned that all "far away carrying actions" occur between times 47 and 48 minutes which are mutually exclusive with times 0 to 28 minutes. Since all shed carrying actions occur between times 0 and 28 minutes, a carrying action cannot be both far away and involve a shed. On the other hand, no mutual exclusivity rules apply for the pair of sub-clusters (1) and (3) - a carrying action may be both far away and involve carrying a house. In fact, since we know that carrying a shed and carrying a house are mutually exclusive, we know that *all* far away actions must involve carrying a house. Thus, we get the sub-cluster hierarchy shown in Figure 7-25.

Drop Action

- If a tornado dropped a shed, then it likely dropped it from an elevation between 0 and 15.5 meters, and vice versa.
- A tornado dropping a shed and a house are mutually exclusive e.g. a tornado drops only one object per drop action.

This set of rules leads to the creation of a sub-cluster hierarchy similar to the one for the carry action, as shown in Figure 7-26. In this case, we learn a new concept that would be called something like "low drop actions", which is a subset of the concept "shed drop actions", since all low drops were of sheds, and not houses. Therefore, we again learn three new concepts for the drop concept cluster using the attribute correlation algorithm.

Destruction Action

- A tornado destroying a shed and a house are mutually exclusive e.g. a tornado destroys only one object per destruction action.
- Only houses were destroyed between times 29 and 35 minutes.

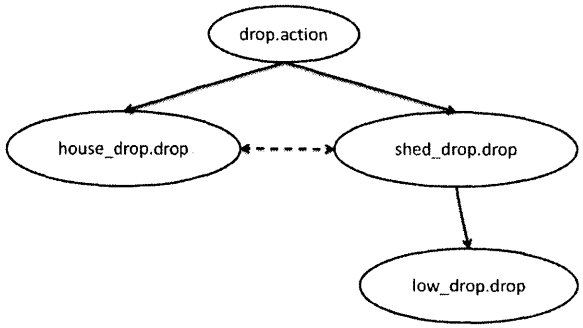

Figure 7-26: Learned Sub-cluster Hierarchy for the Drop Action

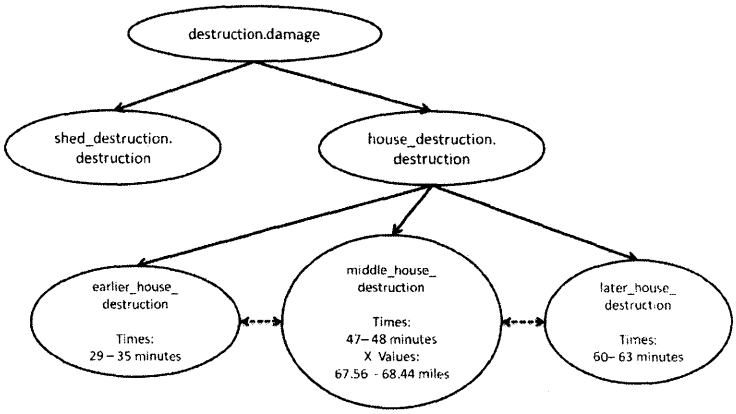

Figure 7-27: Learned Sub-cluster Hierarchy for the Destruction Action

- Only houses were destroyed between times 47 and 48 minutes.
- Only houses were destroyed between times 60 and 63 minutes.
- If destruction occurred between times 47 and 48 minutes, then it took place between x dimension values 67.56 and 68.44 miles from the origin, and vice versa.

This set of rules leads to the creation of the sub-cluster hierarchy shown in Figure 7-27. Thus, we have created five new concepts, which, if we were to name them, might have names such as (1) shed destruction (2) house destruction (3) earlier house destruction (4) middle house destruction (5) late house destruction.

Bystander Observation For the set of nodes that represented observations made by people near the tornado, the attribute correlation algorithm learns the following rule:

- If an observation was made between z dimension locations 102.5 and 104 miles away from the origin, it is likely to observe that a power outage has happened.
- If an observation was made between times 40 and 43 minutes from the start of the simulation, it is likely to take place between z dimension locations 102.5 and 104 miles, and to observe

debris and a power outage.

- If an observation was made between times 45 and 46 minutes, then it likely took place between z dimension locations 102.5 and 104 miles, and to observe a power outage but *no* debris.

As with carry and destruction actions, this allows us to pinpoint the time and area in which this set of observations occurred. Since tornado observations can only be made if the bystander actually *sees* - and is therefore relatively near - a tornado, this rule actually allows us to place the tornado in a particular area during the given time frame. Thus, we know that between times 45 and 46 minutes after the start of the tornado, people saw it passing through the area with z dimension values between 102.5 and 104 miles away from the origin. This is exactly the same type of learning that would allow us, given a dataset of Twitter feeds about tornado sightings, to locate a tornado at a particular time. If we were to adjust the ϵ value to be smaller and smaller, we would obtain correspondingly smaller and more precise time and location ranges, which would ultimately allow us to map out the trajectory of a given tornado based on tornado sighting data points. However, as we would make the value of ϵ smaller, we would run the risk of creating ranges too small to be conceptually meaningful. Since our ultimate goal is to enable our knowledge graph to learn new concepts, we chose larger ϵ values that would enable the representation to learn useful new meanings and ideas.

Tornadoes When the attribute correlation algorithm was applied to tornadoes, it learned the following rules:

- If a tornado dropped an object, it means it also likely carried it, and vice versa.
- If a tornado dropped an object, and therefore, also carried it, it is likely not a weak tornado and vice versa.
- If a tornado dropped an object, and therefore, also carried it, it is likely rated EF3, EF4 or EF5, and vice versa.
- If a tornado has a weak severity, it is likely rated EF0 or EF1.
- If a tornado has a strong severity, it is likely rated EF2 or EF3.
- If a tornado has a violent severity, it is likely rated EF4 or EF5.
- If a tornado has a rating of EF0, it was unlikely to cause fatalities.
- If a tornado has a rating of EF0, it likely damaged buildings, but did not destroy them.
- If a tornado has a rating of EF3, it likely carries and drops objects.

- If a tornado has a rating of EF3, it has a strong severity.
- If a tornado has a rating of EF3, it likely destroyed around 7 buildings.
- If a tornado has a rating of EF4, it likely destroys buildings.
- If a tornado has a rating of EF5, it likely carries and drops objects.
- If a tornado has a rating of EF5, it likely destroys buildings.
- If a tornado destroyed around 2 buildings, it is likely to have a path length between 3.19 and 3.6 miles.
- If a tornado did not destroy any buildings, it likely did not cause fatalities.
- If a tornado has a path length between 0.3 and 0.52 miles, it is likely a weak severity tornado with rating EF0.
- If a tornado has a path length between 1.04 and 1.34 miles, it likely damaged, but did not destroy, buildings.
- If a tornado has a path length between 1.04 and 1.34 miles, it likely did not cause fatalities.
- If a tornado has a path length between 1.04 and 1.34 miles, it is likely has a rating of EF0.
- If a tornado has a path length between 1.04 and 1.34 miles, it likely did not destroyed buildings.
- If a tornado has a path length between 3.19 and 3.6, it likely has a severity EF1.
- If a tornado has a path length between 5.56 and 5.75 miles, it is likely rated EF2.

Thus, using these rules the semantic knowledge representation can learn nine new concepts, arranged in the hierarchy shown in Figure 7-28. This result means that the algorithm was successfully able associate each Enhanced Fujita rating with a tornado severity: weak, strong or violent.

Furthermore, for each new concept, the knowledge representation has learned a typical set of attributes that encompasses how likely it is that each EF rated tornado (1) causes fatalities (2) damages buildings (3) destroys buildings (4) carries objects through the air. In addition, for the tornadoes that contained sufficiently densely clustered data points e.g. tornadoes with severities EF0 - EF2, the attribute correlation algorithm identified a typical path length range as well.

Conversely, for tornadoes with severities EF3 - EF5, the attribute correlation algorithm was unable to identify a specific path length range due to the fact that the path lengths for these tornadoes varied too greatly in size. The same high level of variation prevented the algorithm from associating specific *amounts* of building damage or destruction with tornado severity, since these amounts varied to greatly to be densely clustered.

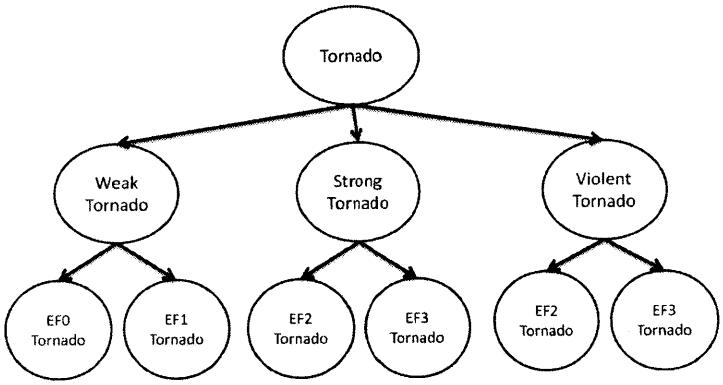

Figure 7-28: Learned Tornado Hierarchy

Conclusion Thus, for 8 out of 22 clusters, the sub-clustering algorithm produced results that allow our semantic knowledge model to adjust itself and create new sub-clusters, and therefore, new concepts. Given 22 types of concepts with 7842 data points, our semantic knowledge model was able to learn 31 new sub-categorizations of these concepts. Furthermore, the algorithm outlined the specific characteristics of each of these 31 new concepts. In other words, the algorithm learned the *definitions* of each of these 31 new concepts. The definitions are both context-specific - as prototype theory dictates they should be - and combine qualitative and quantitative attributes.

Chapter 8

Further Work

This thesis focuses on several algorithms that build a knowledge representation from semantic data and then grow the network by learning new concepts and relationships. The algorithms described are a few methods that can accomplish these tasks. There are many ways that these algorithms can be expanded upon and new ones created to learn in different and potentially more powerful ways. We provide a few suggestions for such areas of improvement.

8.1 Expanding Algorithms to Arbitrarily Shaped Sub-Graphs

Currently, most algorithms proposed in this thesis only compare nodes by examining their immediate, or first level set of attributes. The relational numeric algorithm, for instance, looks for a relationship that is specific to a particular subgraph shape: it looks for a node with two attributes, where both of these attributes have the same numeric component. For instance, given an "above" instance node, the algorithm looks at the two attributes of this node - the two objects being compared - and looks to see if they both have numeric y-coordinates. For this specific type of node, it can detect a consistent "greater than" pattern. Of course, this is not the only possible subgraph for which a consistent numeric relationship can exist. For instance, applying the example of the "above" preposition to another scenario: suppose we have a hot air balloon with people in it that is described as "above" the ground. Not only do we know that the elevation of the hot air balloon is "greater than" that of the ground, we also know that every person riding in the hot air balloon must also have that "greater than" relationship with the ground. Thus, we would want to detect a different shaped subgraph: one in which we look past the first level of attributes for a given node. Since a consistent "greater than" relationship could be possible at *any* level, we would need expand the numeric relational algorithm to have a more complex implementation. Furthermore, we would need this implementation to be very efficient to search through all the possible subgraph shapes.

8.2 Multiple Inheritance

In our knowledge representation, a single node can have multiple parents; however our algorithms do not exploit this fact. Many of the implementation details were decided with this in mind. The choice of the Fuzzy C-Means Clustering Algorithm is one such example, since the use of fuzzy clustering enables a single node to belong to multiple clusters, or concepts. Each concept cluster that a node is placed in corresponds to a different parent. Similarly, many of the helper methods in the Memory class, which maintain and modify the state of the semantic knowledge graph, were implemented to work for a graph in which a single node can have any number of parents. For testing purposes, however, the test datasets were constructed in such a way that each node would only have one parent. In the future, we would hope to explore the implications of multiple inheritance on the performance of our algorithms. It is likely that improvements would have to be made to several of the algorithms to be more efficient for these sorts of datasets.

8.3 Modification of the Semantic Knowledge Graph

Many of the algorithms presented in this paper output results that allow the knowledge graph to learn and grow. In all cases, the algorithms show a high, but not 100% confidence in their results. As such, if we were to make the modifications proposed by the algorithms, there is a non-zero chance that this modification would have to be undone, and re-learned, at a later point. As with any learning algorithm, the semantic learning algorithms are limited by the information given, and thus may make incorrect assumptions given insufficient or inaccurate data. In this case, if an algorithm is provided additional data that improves the overall accuracy and completeness of the dataset, it should be able to re-generate more correct results. In the future, we would like to develop ways for our algorithms to undo incorrect changes that they previously made to the knowledge graph and re-modify the knowledge graph with updated correct results.

8.4 Knowledge Representation to Text Conversion

Currently our algorithms capture knowledge in a machine-readable format which is a combination of text, numbers and symbols. This representation was derived from unstructured natural language and structured numeric data. A very useful function of our knowledge application would be the recreation of text from the knowledge graph. Given this function, cluster prototypes could be converted to textual descriptions - definitions of concepts. This requires the ability to convert from the semantic knowledge graph representation into text, which is an area that would require further work and implementation. This would be an incredibly useful addition to the semantic knowledge application, since it would make it much more comprehensible to users not familiar with the implementation

details of our application.

8.5 Scaling

In choosing and implementing our learning algorithms, we attempted to make them as efficient as possible so that they could be scaled to large datasets. The Fuzzy C-Means Clustering algorithm, for instance, was chosen as the attribute clustering algorithm because it has been shown to perform well on large databases. However, given that our knowledge representation and algorithms are still in development, we have not yet tested them on datasets larger than the natural disaster dataset. In particular, we would eventually like to apply our implementation to datasets that include social networking sites such as Twitter, news stories and online numeric data sources, such as the National Weather Service, which tracks meteorological data. Given these larger real-world datasets, it would be interesting to observe the patterns that could be detected by combining these qualitative and quantitative attributes.

Bibliography

[1] Glossary of meteorology. http://amsglossary.allenpress.com/glossary/search?id=tornado1.

[2] Tornadoes: Nature's most violent storms. http://www.nssl.noaa.gov/edu/safety/tornadoguide.html.

[3] A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 3:32–57, 1974.

[4] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. Technical report, University of Munich, 1996.

[5] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[6] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2005.

[7] Hermann Helbig. Knowledge representation with multilayered extended semantic networks. http://pi7.fernuni-hagen.de/forschung/multinet/multinet_en.html.

[8] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 2008.

[9] Frank Manola and Eric Miller. Rdf primer. Technical report, W3C, 2004.

[10] Elizabeth Millard. Promise of a better internet. *Teradata Magazine*, 10:1, 2010.

[11] Wikipedia. Dbscan. http://en.wikipedia.org/wiki/DBSCAN.

[12] Wikipedia. Multinet. http://en.wikipedia.org/wiki/MultiNet.

[13] Wikipedia. Semantics. http://en.wikipedia.org/wiki/Semantics.

[14] Wikipedia. Tornado. http://en.wikipedia.org/wiki/Tornado.

[15] Wikipedia. Unity game engine. http://en.wikipedia.org/wiki/Unity_(game_engine).